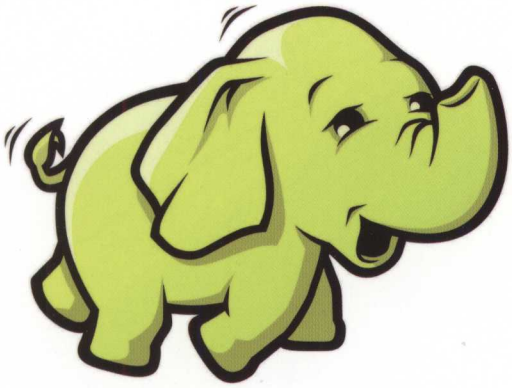
课程 | 实验 | 题库

教育部产学合作协同育人项目成果教材
西普教育研究院 IT 前沿技术方向高校系列教材



# Hadoop 大数据开发
# 案例教程与项目实战

在线实验 + 在线自测

西普教育研究院 时允田 林雪纲 主编
马云涛 薛乔毓 武功成 副主编

实验吧
让实验更简单

-SaaS
开放实验云平台

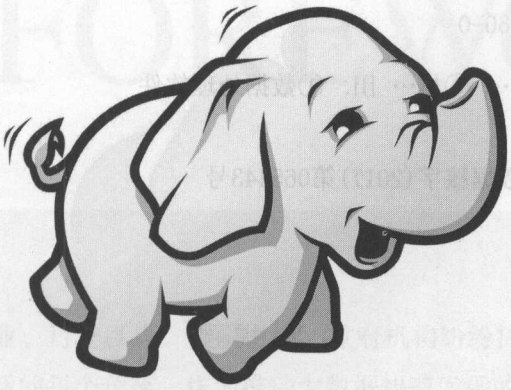
- 内容新颖，可操作性强，层层深入，简明易懂。
- 从实用角度出发，重点培养动手解决问题的能力。
- 提供体系完整的 100 学时在线实验，即学即练，书网结合。
- 包含 96 个案例实战（课程配套案例 36 个、扩展案例 60 个）和 50 余套自测题。
- 附赠全部案例源代码和操作视频等资源。

扫描书中二维码随时进行在线测试

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

-SaaS 课程 | 实验 | 题库

教育部产学合作协同育人项目成果教材
西普教育研究院 IT 前沿技术方向高校系列教材



# Hadoop大数据开发
# 案例教程与项目实战

在线实验 + 在线自测

西普教育研究院 时允田 林雪纲 主编
马云涛 薛乔毓 武功成 副主编

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Hadoop大数据开发案例教程与项目实战 ： 在线实验+在线自测 / 时允田，林雪纲主编. -- 北京 ： 人民邮电出版社，2017.5
ISBN 978-7-115-45360-0

Ⅰ. ①H… Ⅱ. ①时… ②林… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2017)第065443号

## 内 容 提 要

本书是一本 Hadoop 学习入门参考书，全书共 11 章，分为基础篇和提高篇两部分。基础篇包括第 1~6 章，具体包括 Hadoop 概述、Hadoop 基础环境配置、分布式存储 HDFS、计算系统 MapReduce、计算模型 Yarn、数据云盘。提高篇包括第 7~11 章，具体包括协调系统 Zookeeper、Hadoop 数据库 Hbase、Hadoop 数据仓库 Hive、Hadoop 数据采集 Flume、OTA 离线数据分析平台。全书内容结构合理，知识点全面，讲解详细，重点难点突出。

本书适合作为院校计算机及相关专业大数据课程的教材，也可供学习者自学参考。

◆ 主　　编　西普教育研究院　时允田　林雪纲
副 主 编　马云涛　薛乔毓　武功成
责任编辑　左仲海
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京市昌平百善印刷厂印刷

◆ 开本：787×1092　1/16
印张：18.25　　2017 年 5 月第 1 版
字数：425 千字　　2017 年 5 月北京第 1 次印刷

定价：49.80 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字20170147号

# 前 言 FOREWORD

## 编写目的

现今各大电商企业、行业巨头、科研机构、政府机构纷纷提出向大数据进军，大数据逐渐充满我们生活的每个角落，作为推动大数据快速发展的 Hadoop 产品自然受到众多企业和开发者的欢迎。从事 Hadoop 开发的人员越来越多，应用 Hadoop 产品的企业也越来越多，大数据相应课程也逐渐成为高校计算机相关专业必修的课程之一。

## 平台支撑

为了让广大学习者能够快速入门，本书以实践案例为主线，通过遵循书中的案例操作步骤，完成一个个实验案例，学习 Hadoop 开发技术。同时，北京西普阳光教育科技股份有限公司（简称西普教育）开发的在线教育平台——实验吧（http://www.shiyanbar.com），提供了强大的集成实验环境及海量的在线教学资源，把本书配套的实验搬到线上，可以让读者更方便地结合本书进行动手实践。

1. 如何学习本书中配套实验课程

（1）购买本书后，找到粘贴在本书封底的刮刮卡，刮开获得学号。

（2）登录实验吧网站（www.shiyanbar.com），完成网站注册。

（3）登录人邮学院在线实验中心（rymooc.shiyanbar.com），输入在实验吧注册的账户及密码，完成登录（见图 1）。

（4）输入刮刮卡中的学号，姓名填写“人邮学院”，单击保存，完成绑定（见图 2）。


图 1　登录在线实验平台

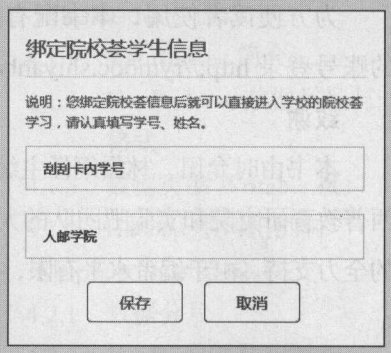



图 2　绑定学生信息

（5）完成绑定后，自动登录进入在线实验中心，开始学习本书配套课程资源。

2．如何学习本书中配套练习题

实验吧教研团队为本书配套了丰富的课后练习题，并提供正确答案，读者通过扫描本书章节里配套的习题二维码，即可进行在线自测，提交后自动判断正误（见图 3）。

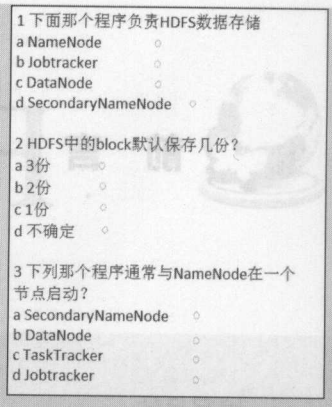



图 3　在线测试

**本书特点**

本书作者团队有着多年的大数据实际项目开发经验，也有丰富的高校教学和 IT 培训经验。作者带领西普教育研究院的云计算及大数据师资团队在培训机构，南京大学、大连理工大学等高校专业课程教学和学生培训过程中进行了多次基于案例化教学、在线实验、翻转课堂的创新教学实践，本书也是系列教学实践的成果之一。

本书主要特点如下。

1．真正以案例为驱动

为了使读者能快速地掌握相关技术，本书各章都以实际案例来驱动学习，让读者真正地实现边实操边学习。

2．合理、高效的组织结构

本书采用“做、学、教一体化”的教学方法，培养读者的主观能动性，在教与学的过程中弱化“教”，深化“做”与“学”，最终达到“老师轻松教，学生高效学，实验简单做”的目的。

3．内容线下线上同步，丰富实用

本书的训练紧紧围绕着与实际相结合的实验进行，各章做简要的知识点分析后，读者可以随时随地在线进行对应的实验，免去了读者配置开发环境的困扰。

**资源下载**

为方便读者使用，本书配有全部实例的源代码及电子教案。读者可通过本书配送的账号登录 http://rymooc.shiyanbar.com/网站进行资料下载与在线学习。

**致谢**

本书由时允田、林雪纲任主编，马云涛、薛乔毓、武功成任副主编，编写过程中得到西普教育研究院和实验吧团队的大力协助，特别感谢吴新国、肖何、陈玉泉、芦治国等人的全力支持。由于编者水平有限，书中不妥或错误之处在所难免，期望广大读者批评指正。

编者

2017 年 1 月

# 目 录 CONTENTS

## 基 础 篇

# 提 高 篇

# 基 础 篇

# 第 1 章 Hadoop 概述

## 本章要点

- Hadoop 简介。
- 了解 Hadoop 的相关项目。
- 了解 Hadoop 的来源和发展。
- 了解 Hadoop 的特点和体系架构。

## 引　言

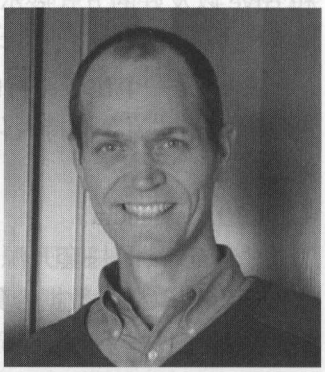


图 1-1　道格 · 卡丁图

继云计算技术之后，大数据时代快速来临，大数据充满世界的每个角落，发展势头盖过任何一门技术。以 Hadoop 为首的大数据平台为大数据掀起一阵狂潮，也让全世界认识了道格 · 卡丁（见图 1-1）以及大数据带给人类的贡献。并随着世界上的 IT 巨头企业分别研发出同类大数据平台，再次把大数据应用推上高潮。本章通过对 Hadoop 的基本知识、Hadoop 来源、Hadoop 发展史和特点、Hadoop 体系架构以及 HDFS 体系结构的介绍，让读者对 Hadoop 有一个初步的认识。

## 1.1 Hadoop 简介

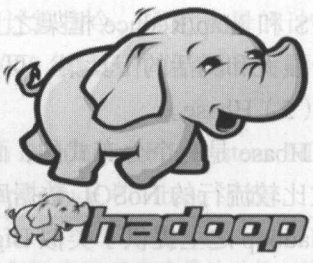



图 1-2　Logo

Hadoop 是一个由 Apache 基金会开发的开源软件，具有可靠性、扩展性的分布式的计算存储系统，标识性 Logo 为一个黄色小象（见图 1-2）。Hadoop 软件库作为一个框架，它可以轻松地通过 1 台到数千台服务器联合在一起实现对大数据进行存储和计算，而且每一个都能提供存储和计算能力。用户可以在不了解 Hadoop 底层细节的情况下，开发分布式程序，能够十分方便地利用集群的强大能力进行程序运算，而且能够解决高可用问题。Hadoop 系统实现了一个分布式文件系统（Hadoop Distributed File System，HDFS）。HDFS 有高容错性的特点，并且设计用来部署在低廉的硬件上。它提供高吞吐量（high throughput）

来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。

Hadoop 的框架核心的设计是 HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，而 MapReduce 为海量的数据提供了计算。

Hadoop 项目主要包括 4 个部分。

- Hadoop Common:支撑其他模块。
- Hadoop Distributed File System：分布式系统对应用提供高吞吐量的访问。
- Hadoop Yarn：资源管理和任务调度的一个框架。
- Hadoop MapReduce：能够并行处理大数据集的 Yarn 基本系统。

## 1.2 Hadoop 相关项目

在 Apache 项目中和 Hadoop 相关联的项目有很多，常见的项目包括：

（1）Ambari。

Apache Ambari 是一种基于 Web 的工具，支持 Apache Hadoop 集群的供应、管理和监控。Ambari 目前已支持大多数 Hadoop 组件，包括 HDFS、MapReduce、Hive、Pig、Hbase、Zookeeper、Sqoop 和 Hcatalog 等。Ambari 也提供了一种仪表盘用来查看集群健康状况，Pig 和 Hive 以友好的方式展示特有的特征。

（2）Avro。

Avro 是一个比较流行的数据序列化系统，可以提供丰富的结构类型，快速可压缩的二进制数据格式，存储持久化数据，支持远程过程调用协议（Remote Procedure Call Protocol，RPC）。

（3）Cassandra。

Cassandra 是一套开源分布式 NoSQL 数据库系统。它最初由 Facebook 开发，用于储存收件箱等简单格式数据，集合 Google BigTable 的数据模型与 Amazon Dynamo 的完全分布式的架构于一身。Facebook 于 2008 年将 Cassandra 开源，此后，由于 Cassandra 良好的可扩展性，被 Digg、Twitter 等知名 Web 2.0 网站所采纳，成为了一种流行的分布式结构化数据的存储方案。

（4）Chukwa。

Chukwa 是一个开源的用于监控大型分布式系统的数据收集系统。它构建在 Hadoop 的 HDFS 和 Map/Reduce 框架之上，继承了 Hadoop 的可伸缩性和健壮性。Chukwa 还包含了一个强大和灵活的工具集，可用于展示、监控和分析已收集的数据。

（5）Hbase。

Hbase 是一个分布式的，面向列的开源数据库，可以称为 Hadoop 的标准数据库，也是一款比较流行的 NoSQL 数据库，由 Google 公司发表的论文 BigTable 经过演变而来，Hbase 在 Hadoop 之上提供了类似 BigTable 的能力，主要解决非关系型数据存储问题。

（6）Hive。

Hive 本身是建立在 Hadoop 体系结构上的数据仓库基础构架，可以将结构化的数据文件映射为一张数据库表，并提供完整的查询语言（QueryLanguage，QL）语句，把 QL 语句转化成 MapReduce 程序提交给 Hadoop 集群完成相关任务。它提供了一系列的工具，可以

用来进行数据提取转化加载（Extract-Transform-Load，ETL），这是一种可以存储、查询和分析并存储在Hadoop中的大规模数据处理的机制。

（7）Mahout。

Mahout是阿帕奇软件基金会（Apache Software Foundation，ASF）旗下的一个开源项目，提供一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。Mahout包含许多实现，包括聚类、分类、推荐过滤、频繁子项挖掘。此外，通过使用Apache Hadoop库，Mahout可以有效地扩展到云中。

（8）Pig。

Pig是一个基于Hadoop的大规模数据分析平台，它提供的SQL-LIKE语言叫Pig Latin，该语言的编译器会把类SQL的数据分析请求转换为一系列经过优化处理的MapReduce运算。Pig为复杂的海量数据并行计算提供了一个简单的操作和编程接口。

（9）Spark。

Apache Spark是一个快速和通用的集群计算系统。它还支持一组丰富的高级工具，包括Spark SQL、SQL和结构化数据处理、MLlib机器学习、GraphX图形处理、Spark流。

（10）Zookeeper。

Zookeeper是一个能够高效开发和维护分布式的开放源码的应用协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括维护配置信息、名字服务、分布式同步、组服务等。这些服务都被应用在分布式应用程序或其他一些形式。

## 1.3 Hadoop来源

Hadoop这个名字不是常见的几个单词的缩写，而是由道格·卡丁虚构的一个名字。道格·卡丁曾解释Hadoop的由来："这个名字是我孩子给一个棕黄色的大象玩具起的名。命名的标准就是简短，容易发音和拼写，没有太多的意义，并且不会被用于别处。小孩子恰恰是这方面的高手"。道格·卡丁是Lucene、Nutch 、Hadoop等项目的发起人。

Hadoop的出现来自Google的两款产品：GFS和MapReduce。GFS用于存储不同设备所产生的海量数据，可以解决在网络抓取和索引过程中产生的大文件存储问题。MapReduce运行在GFS之上，负责分布式大数据的计算，可以处理海量网页的索引问题。MapReduce框架解决问题的思路就是把一个应用程序分解为许多并行的计算指令，通过大量的计算节点运行指令并产生非常巨大的数据集。后来由ASF公司于2005年秋天作为Lucene的子项目Nutch的一部分正式引入。

2006年3月份，Map/Reduce和Nutch分布式文件系统（Nutch Distributed File System，NDFS）分别被纳入称为Hadoop的项目中。Hadoop主要由HDFS、MapReduce和Hbase组成。

学习小贴士

道格·卡丁，1985年毕业于美国斯坦福大学，大学期间对IT产生浓厚的兴趣，第一份工作是在Xerox实习并开发屏幕保护平台程序。1997年，道格·卡丁发布Lucene。2004年，道格·卡丁发布Nutch。2006年受邀加入Yahoo。后来又加入Cloudera。

## 1.4 Hadoop 的发展史

2002 年，Hadoop 起始于 Apache 项目的 Nutch。

2003 年，Google 发布关于 GFS 的论文。

2004 年，Nutch 的开发者开发了 NDFS。

2004 年，Google 发表了关于 MapReduce 的论文。

2004 年，由道格·卡丁开发了现在 HDFS 和 MapReduce 的最初版本。

2005 年，MapReduce 被引入 NDFS。

2005 年 12 月，Nutch 移植到新框架，Hadoop 在 20 个节点上稳定运行。

2006 年 1 月，道格·卡丁加入 Yahoo!。

2006 年 2 月，Apache Hadoop 项目正式启动以支持 MapReduce 和 HDFS 的独立发展。

2006 年 2 月，Yahoo！的网格计算团队采用 Hadoop。

2006 年 4 月，在 188 个节点上（每个节点 10 GB）运行排序测试需要 47.9 小时。

2006 年 5 月，Yahoo！建立了一个 300 个节点的 Hadoop 研究集群。

2006 年 5 月，在 500 个节点上运行排序测试需要 42 小时（硬件配置比 4 月的更好）。

2006 年 11 月，研究集群增加到 600 个节点。

2006 年 12 月，排序测试集在 20 个节点上运行 1.8 小时，100 个节点上运行 3.3 小时，500 个节点上运行 5.2 小时，900 个节点上运行 7.8 小时。

2007 年 1 月，研究集群增加到 900 个节点。

2007 年 4 月，研究集群增加到两个 1000 个节点的集群。

2008 年 4 月，在 900 个节点上运行 1 TB 排序测试仅需 209 秒，成为世界最快。

2008 年 10 月，研究集群每天装载 10 TB 的数据。

2009 年 3 月，17 个集群总共 24 000 台机器。

2009 年 4 月，赢得每分钟排序，59 秒内排序 500 GB（在 1400 个节点上）和 173 分钟内排序 100 TB 数据（在 3400 个节点上）。

2009 年 7 月，Avro 和 Chukwa 成为 Hadoop 新的子项目。

2010 年 5 月，Avro 脱离 Hadoop 项目，成为 Apache 顶级项目。

2010 年 5 月，Hbase 脱离 Hadoop 项目，成为 Apache 顶级项目。

2010 年 5 月，IBM 提供了基于 Hadoop 的大数据分析软件——InfoSphere BigInsights，包括基础版和企业版。

2010 年 9 月，Hive（Facebook）脱离 Hadoop，成为 Apache 顶级项目。

2010 年 9 月，Pig 脱离 Hadoop，成为 Apache 顶级项目。

2011 年 1 月，ZooKeeper 脱离 Hadoop，成为 Apache 顶级项目。

2011 年 3 月，Apache Hadoop 获得 Media Guardian Innovation Awards。

2011 年 3 月，Platform Computing 宣布在它的 Symphony 软件中支持 Hadoop MapReduce API。

2011 年 5 月，HCatalog 1.0 发布。该项目由 Hortonworks 在 2010 年 3 月份提出，HCatalog 主要用于解决数据存储、元数据的问题，主要解决 HDFS 的瓶颈，它提供了一个地方来存

储数据的状态信息，这使得数据清理和归档工具可以很容易地进行处理。

2011 年 4 月，SGI( Silicon Graphics International )基于 SGI Rackable 和 CloudRack 服务器产品线提供 Hadoop 优化的解决方案。

2011 年 5 月，EMC 为客户推出一种新的基于开源 Hadoop 解决方案的数据中心设备——GreenPlum HD，以助其满足客户日益增长的数据分析需求并加快利用开源数据分析软件。Greenplum 是 EMC 在 2010 年 7 月收购的一家开源数据仓库公司。

2011 年 5 月，在收购了 Engenio 之后，NetApp 推出与 Hadoop 应用结合的产品 E5400 存储系统。

2011 年 6 月，Calxeda 公司(之前公司的名字是 Smooth-Stone)发起了“开拓者行动”，一个由 10 家软件公司组成的团队将为基于 Calxeda 即将推出的 ARM 系统上芯片设计的服务器提供支持。并为 Hadoop 提供低功耗服务器技术。

2011 年 6 月，数据集成供应商 Informatica 发布了其旗舰产品，产品设计初衷是处理当今事务和社会媒体所产生的海量数据，同时支持 Hadoop。

2011 年 7 月，Yahoo!和硅谷风险投资公司 Benchmark Capital 创建了 Hortonworks 公司，旨在让 Hadoop 更加鲁棒(可靠)，并让企业用户更容易安装、管理和使用 Hadoop。

2011 年 8 月，Cloudera 公布了一项有益于合作伙伴生态系统的计划——创建一个生态系统，以便硬件供应商、软件供应商以及系统集成商可以一起探索如何使用 Hadoop 更好地洞察数据。

2011 年 8 月，Dell 与 Cloudera 联合推出 Hadoop 解决方案——Cloudera Enterprise。Cloudera Enterprise 基于 Dell PowerEdge C2100 机架服务器以及 Dell PowerConnect 6248 以太网交换机。

2013 年 5 月，Hadoop 与 Cloudera 完全整合。

2014 年 2 月，Hadoop 发布 Hadoop 2.3.0。

2014 年 4 月，Hadoop 发布 Hadoop 2.4.0。

2014 年 8 月，Hadoop 发布 Hadoop 2.5.0。

2014 年 11 月，Hadoop 发布 Hadoop 2.6.0。

2015 年 4 月，Hadoop 发布 Hadoop 2.7.0。

2016 年 9 月，Hadoop 发布 Hadoop 3.0.0。

## 1.5 Hadoop 特点

Hadoop 作为比较流行的分布式开源项目系统，提供了存储和处理海量数据的能力，很多大公司，如 Google、Facebook 等都争先使用 Hadoop 作为公司内部产品的技术支撑。Hadoop 所具有的几个特征如下。

（1）高可扩展性。

Hadoop 是一个高度可扩展的存储平台，可以存储和分发横跨数百个并行操作的廉价的服务器数据集群。能可靠地（reliably）存储和处理拍字节（PB）数据。不同于传统的关系型数据库系统不能扩展到处理大量的数据，Hadoop 是能给企业提供涉及成百上千 TB 的数据节点上运行的应用程序。

（2）成本效益良好。

Hadoop 为企业用户提供了极具成本效益的存储解决方案。传统的关系型数据库管理系统的问题是不符合海量数据的处理器，不能够符合企业的成本效益。Hadoop 的架构则不同，其被设计为一个向外扩展的架构，可以经济地存储所有公司的数据供以后使用，节省的费用是非常惊人的，Hadoop 提供数百 TB 的存储和计算能力。

可以通过普通机器组成的服务器群来分发以及处理任务数据。这些服务器群总计可达数千个节点，甚至更多。与一体机、商用数据仓库相比，Hadoop 是开源的，项目的软件成本因此会大大降低

（3）灵活性更好。

Hadoop 能够帮助企业轻松地访问数据源，并可以分析不同类型的数据，从这些数据中产生价值，这意味着企业可以利用 Hadoop 的灵活性从社交媒体、电子邮件或单击流量等数据源获得宝贵的商业价值。

此外，Hadoop 的用途非常广，诸如对数处理、推荐系统、数据仓库、市场活动分析以及欺诈检测。

（4）Hadoop 处理更快。

Hadoop 拥有独特的存储方式，用于数据处理的工具通常在与数据相同的服务器上，从而能够更快地处理数据，如果你正在处理大量的非结构化数据，Hadoop 能够有效地在几分钟内处理 TB 级的数据。

通过分发数据，Hadoop 可以在数据所在的节点上并行地（Parallel）处理，这使得处理非常快速高效。

（5）容错能力强。

使用 Hadoop 的一个关键优势就是具有很强的容错能力。当数据被发送到一个单独的节点，该数据也被复制到集群的其他节点上，这意味着在故障情况下，存在另一个副本可供使用。

Hadoop 能自动地维护数据的多份副本，一般默认备份为 3 份，一旦某个节点上的数据损坏或丢失，立刻将失败的任务重新分配。并且在任务失败后能自动地重新部署（Redeploy）计算任务。

## 1.6 Hadoop 体系架构

Hadoop 是实现了分布并行处理任务的系统框架，Hadoop 的核心组成是 HDFS 和 MapReduce 两个子系统，能够自动完成大任务计算和大数据存储的分割工作。随着 Hadoop 近几年的发展，现在 Hadoop 已经包括很多项目，可以称为 Hadoop 的子集。很多 Hadoop 相关的生态项目也应运而生，例如 Common、Yarn、Avro、Chukwa、Hive、Hbase、Zookeeper 等。这些生态工具对 Hadoop 的核心起到了良好的补充作用。

HDFS 系统是 Hadoop 的存储系统，能够实现创建文件、删除文件、移动文件等功能，操作的数据主要是要处理的原始数据以及计算过程中的中间数据，实现高吞吐率的数据读写。MapReduce 系统是一个分布式计算框架，主要任务就是能够利用廉价的计算机对海量数据进行分解处理。

### 1.6.1 HDFS 体系结构

HDFS 和 MapReduce 是 Hadoop 的两大核心。而实际上 Hadoop 的体系结构主要是通过 HDFS 来实现对分布式存储的底层支持的，并且它会通过 MapReduce 来实现对分布式并行任务处理的程序支持。

HDFS 的体系结构，采用主从（Master/Slave）结构模型（见图 1-3），一个 HDFS 集群是由一个 NameNode 若干个 DataNode 组成的。其中 NameNode 作为主服务器，管理文件系统的命名空间和客户端对文件的访问操作。集群中的 DataNode 管理存储的数据。当文件进行存储时，文件会被分割成若干个数据块，并且这些数据块会被存放在集群中的各个 DataNode 节点上。NameNode 执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录等，负责数据块到具体 DataNode 上的映射。DataNode 负责处理文件系统客户端的文件读写请求，并在 NameNode 的统一调度下进行数据块的创建、删除和复制工作。NameNode 和 DataNode 都被设计成可以在普通计算机上运行。这些计算机通常运行的是 GNU/Linux 操作系统。一个典型的部署场景是集群中的一台机器运行一个 NameNode 实例，其他机器分别运行一个 DataNode 实例。NameNode 是所有 HDFS 元数据的管理者，用户数据永远不会存储到 NameNode 上。

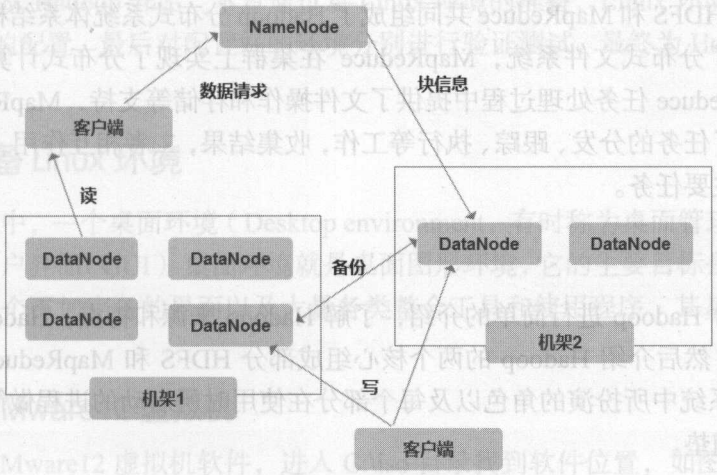



图 1-3 HDFS 的体系结构

### 1.6.2 MapReduce 体系结构

MapReduce 是一种并行编程模式，这种模式使得软件开发者可以轻松地编写出分布式并行程序。在 Hadoop 的体系结构中，MapReduce 是一个简单易用的软件框架，基于它可以将任务分发到由上千台商用机器组成的集群上，并以一种高容错的方式并行处理大量的数据集，实现 Hadoop 的并行任务处理功能。在早期的 MapReduce 框架中，主要是由一个单独运行在主节点上的 JobTracker 进程和运行在每个集群从节点上的 TaskTracker 进程共同组成的（见图 1-4）。主节点 JobTracker 负责调度构成一个作业的所有任务，这些任务分布在不同的从节点 TaskTracker 上。主节点通过心跳机制(心跳,主从节点的通信时间间隔)监控它们的执行情况，并且重新执行之前失败的任务；从节点仅负责由主节点指派的任务。当

一个 Job 被 Client 提交时，JobTracker 接收到提交作业和配置信息之后，就会将配置信息等分发给从节点，同时调度任务并监控 TaskTracker 的执行。后来 MapReduce 的体系结构略有变化，实际上原理还是一致的。

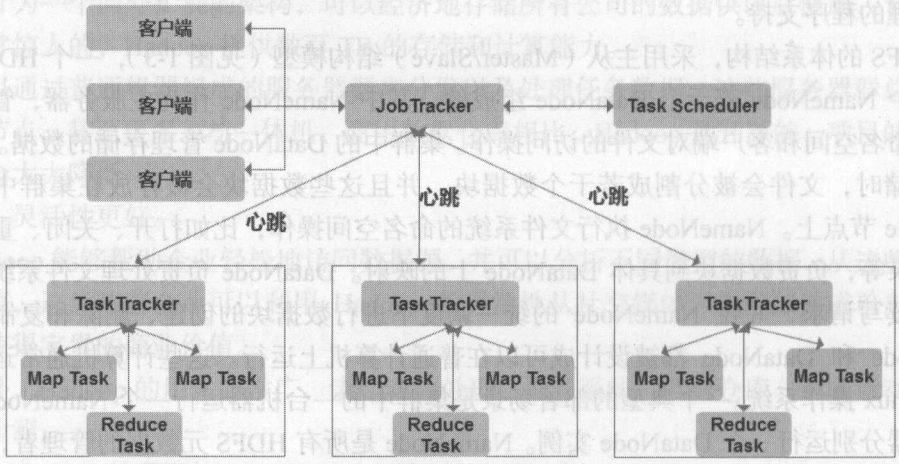



图 1-4　MapReduce 体系结构

也就是说，HDFS 和 MapReduce 共同组成了 Hadoop 分布式系统体系结构的核心。HDFS 在集群上实现了分布式文件系统，MapReduce 在集群上实现了分布式计算和任务处理。HDFS 在 MapReduce 任务处理过程中提供了文件操作和存储等支持，MapReduce 在 HDFS 的基础上实现了任务的分发、跟踪、执行等工作，收集结果，二者相互作用，完成了 Hadoop 分布式集群的主要任务。

## 本章小结

本章首先对 Hadoop 进行简单的介绍，了解 Hadoop 来源和特点，Hadoop 的体系架构的组成及原理。然后介绍 Hadoop 的两个核心组成部分 HDFS 和 MapReduce，并对这两部分在 Hadoop 系统中所扮演的角色以及每个部分在使用时所启动的进程做简单的介绍，为后续内容做好铺垫。

## 习题

1. 简述 Hadoop 平台的发展过程。
2. 简述 Hadoop 名称及技术来源。
3. 简述 Hadoop 的体系架构。
4. 简述 MapReduce 的体系架构。
5. 简述 HDFS 和 MapReduce 在 Hadoop 中的角色。

扫一扫在线测

# 第 2 章 Hadoop 基础环境配置

## 本章要点

- 熟悉 Linux 环境准备。
- 掌握 Linux 系统配置。
- 熟悉 Hadoop 环境搭建。

## 引　言

Hadoop 是大数据发展的领军平台，应该说 Hadoop 把大数据技术推上高潮，所以要学习大数据，应该从 Hadoop 开始。本章通过对 Linux 环境的准备，Linux 环境的配置，以及对 Hadoop 环境的配置，最后对配置好的环境分别进行验证测试，最终为 Hadoop 的开发学习做好一切准备。

## 2.1 准备 Linux 环境

在图形计算中，一个桌面环境（Desktop environment，有时称为桌面管理器）为计算机提供一个图形用户界面（GUI）。桌面环境就是桌面图形环境，它的主要目标是为 Linux/Unix 操作系统提供一个更加完备的界面以及大量各类整合工具和使用程序，其基本易用性吸引着大量的新用户。

### 2.1.1 安装 VMware12 虚拟机

（1）安装 VMware12 虚拟机软件，进入 C:\iso 目录找到软件位置，如图 2-1 所示。

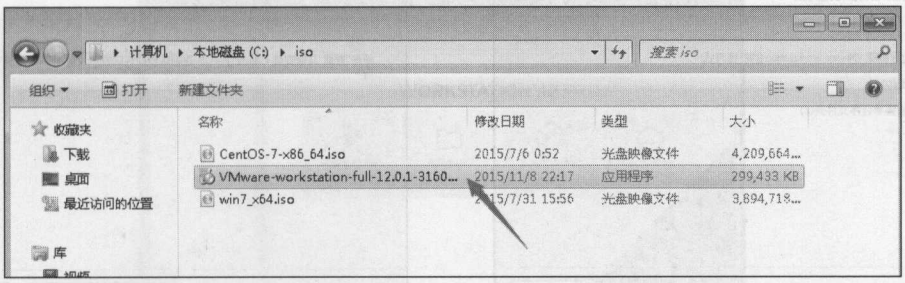



图 2-1　软件位置

（2）双击软件进入安装界面，如图 2-2 所示。

（3）选中“我接受许可协议中的条款”选项后，单击“下一步”按钮，如图 2-3 所示。

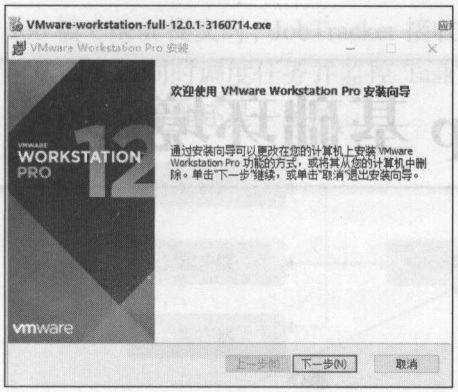



图 2-2　安装界面

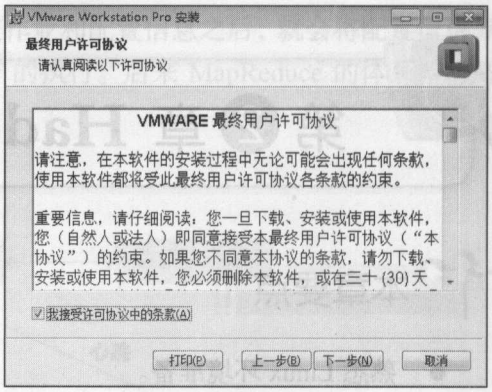



图 2-3　用户许可协议

（4）默认安装位置，单击“下一步”按钮，如图 2-4 所示。

（5）取消勾选“启动时检查产品更新”选项和“帮助完善 VMware Workstation Pro（H）”选项，单击“下一步”按钮，如图 2-5 所示。

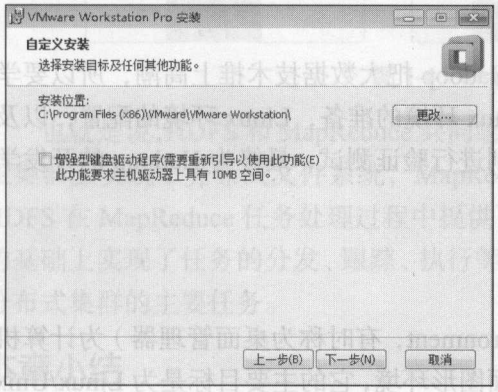



图 2-4　自定义安装

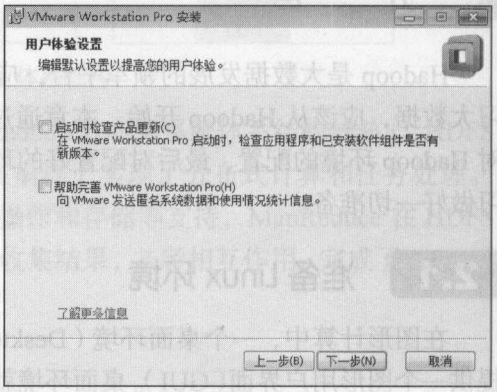



图 2-5　用户体验设置

（6）确定建立快捷方式后单击“下一步”按钮，如图 2-6 所示。

（7）基本设定完成后单击“安装”按钮，开始安装该软件，如图 2-7 所示。

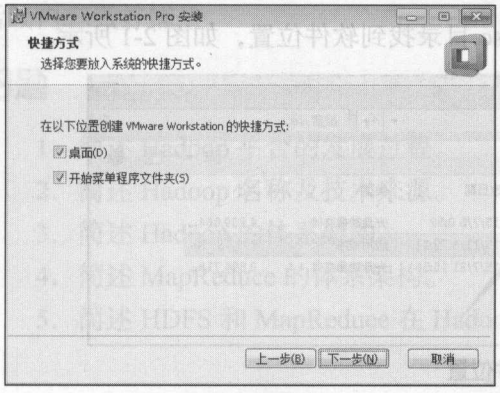



图 2-6　快捷方式

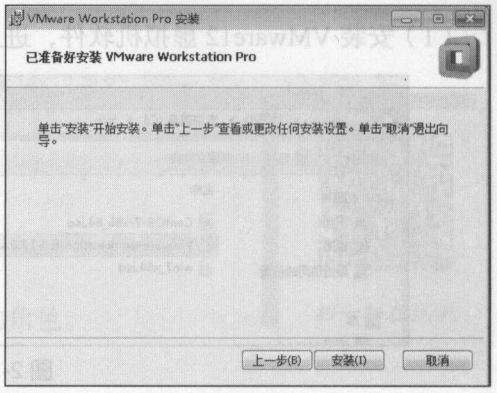



图 2-7　VMware 安装

（8）安装完成后单击“完成”按钮结束安装，如图 2-8 所示。

（9）找到桌面上的虚拟机图标并双击，如图 2-9 所示。

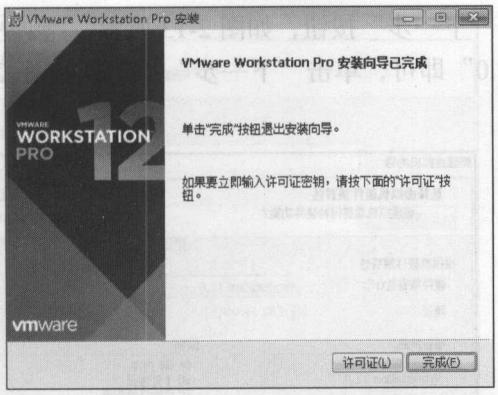



图 2-8 完成安装

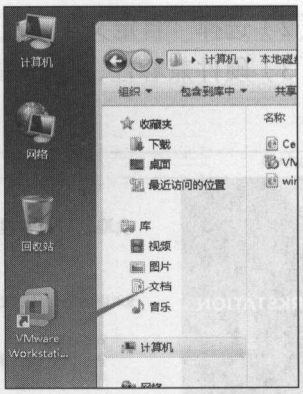



图 2-9 双击虚拟机图标

（10）出现激活使用界面后选择“我希望试用 VMware Workstation 12 30 天（W）”，并输入一个邮箱地址，单击“继续”按钮，如图 2-10 所示。

（11）最终弹出完成界面，单击“完成”按钮，如图 2-11 所示。


图 2-10 使用界面

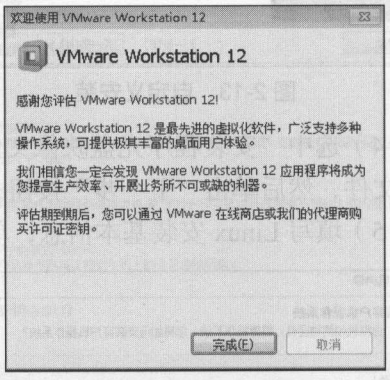



图 2-11 完成安装

## 2.1.2 部署 CentOS 64 位操作系统

（1）找到桌面上的虚拟机图标，双击后，启动 VMware 界面，选择“创建新的虚拟机”，如图 2-12 所示，即会弹出向导。

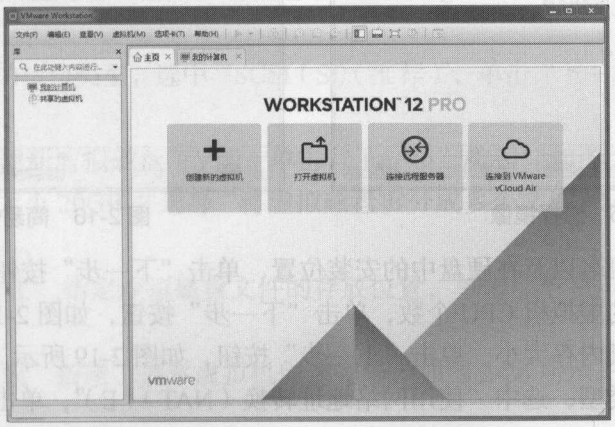



图 2-12 VMware 界面

（2）选择“自定义（高级）（C）”并单击“下一步”按钮，如图 2-13 所示。

（3）硬件兼容性，选择“Workstation 12.0”即可，单击“下一步”按钮，如图 2-14 所示。

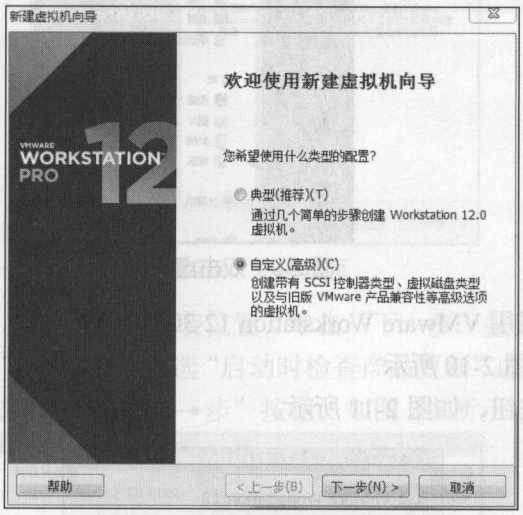



图 2-13　自定义安装

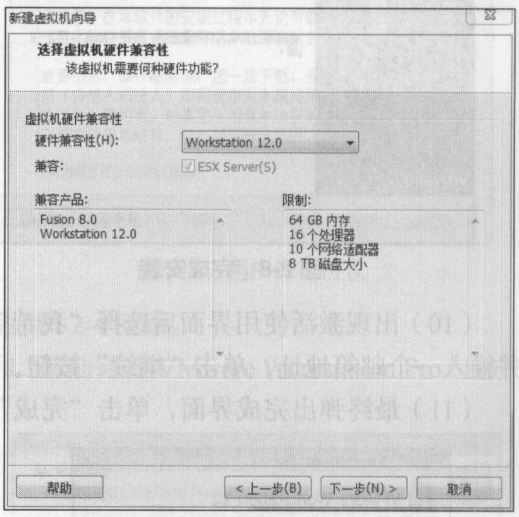



图 2-14　虚拟机向导

（4）选中“安装程序光盘映像文件（iso）（M）”，单击“浏览（R）”按钮,找到对应的映像文件，然后单击“下一步”按钮，如图 2-15 所示。

（5）填写 Linux 安装基本信息，单击“下一步”按钮，如图 2-16 所示。

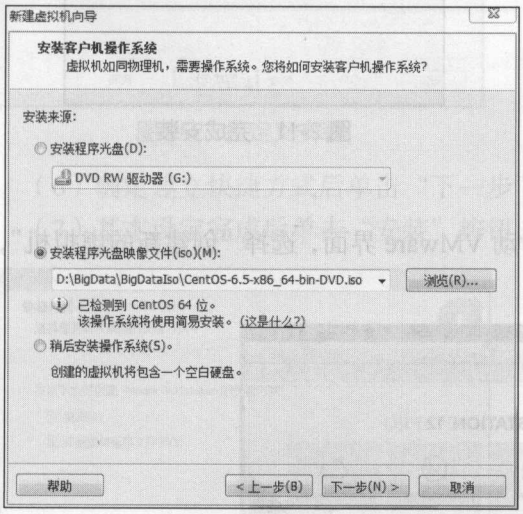



图 2-15　选择镜像

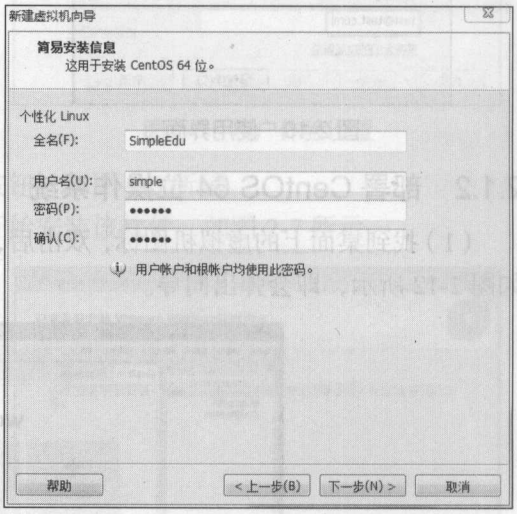



图 2-16　简易安装信息

（6）指定虚拟机名以及在硬盘中的安装位置，单击“下一步”按钮，如图 2-17 所示。

（7）指定建立的虚拟机 CPU 个数，单击“下一步”按钮，如图 2-18 所示。

（8）指定虚拟机内存大小，单击“下一步”按钮，如图 2-19 所示。

（9）指定网络类型，选中“使用网络地址转换（NAT）（E）”，单击“下一步”按钮，如图 2-20 所示。

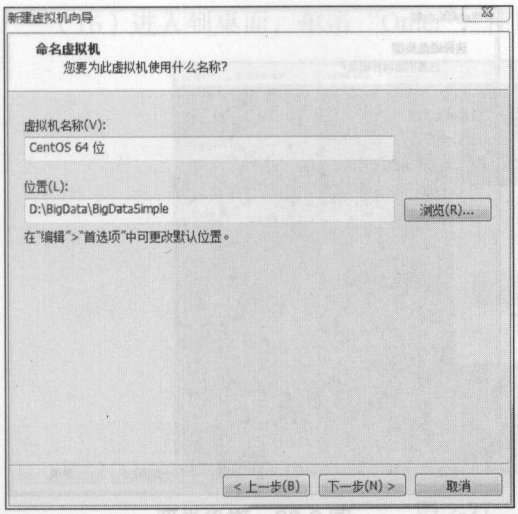



图 2-17 命名虚拟机

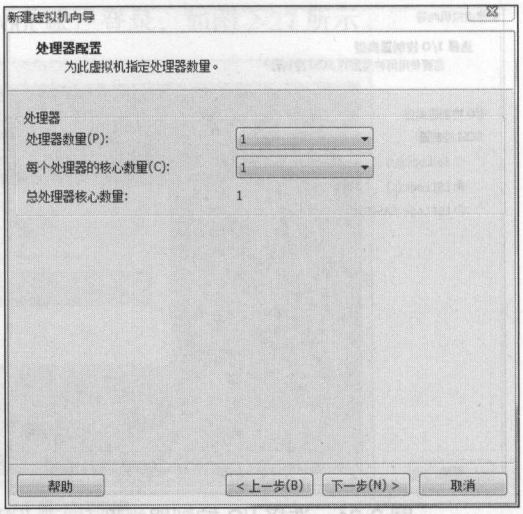



图 2-18 处理器配置

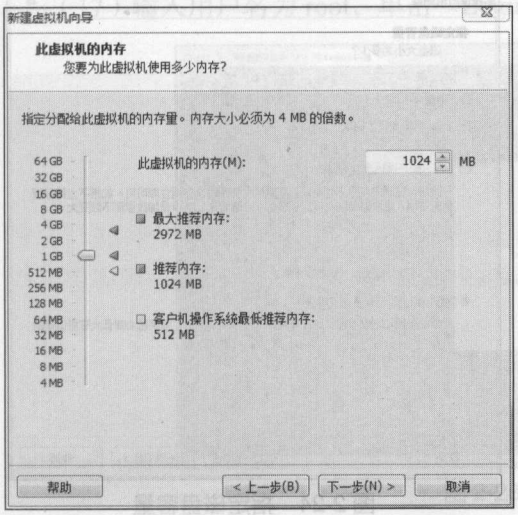



图 2-19 虚拟机内存

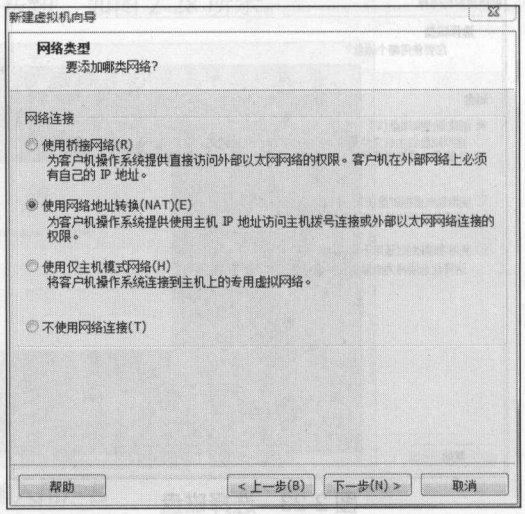



图 2-20 网络类型

（10）指定虚拟磁盘的输入输出控制，选中“LSI Logic（L）”，单击“下一步”按钮，如图 2-21 所示。

（11）指定磁盘的接口类型，选中“SCSI（S）（推荐）”，单击“下一步”按钮，如图 2-22 所示。

（12）选中“创建新虚拟磁盘（V）”，单击“下一步”按钮，如图 2-23 所示。

（13）填写硬盘大小 20GB 并选择“将虚拟磁盘拆分成多个文件（M）”，单击“下一步”按钮，如图 2-24 所示。

（14）单击“浏览”,指定虚拟磁盘文件的存放位置，单击“下一步”按钮，如图 2-25 所示。

（15）单击“完成”按钮，表示虚拟机创建完毕，如图 2-26 所示。

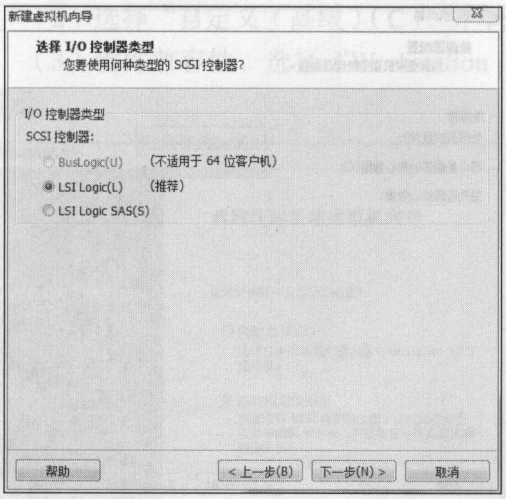



图 2-21 选择 I/O 控制器类型

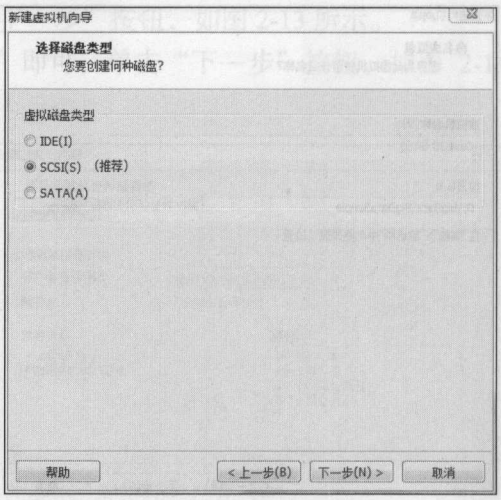



图 2-22 磁盘类型

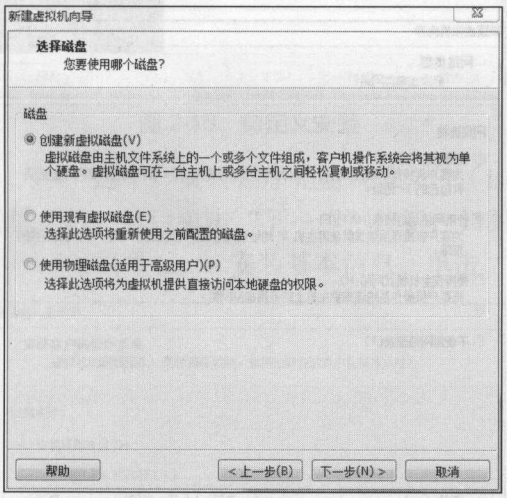



图 2-23 选择磁盘

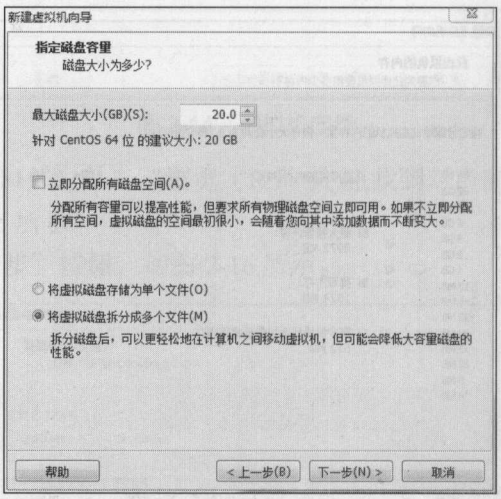



图 2-24 指定磁盘容量

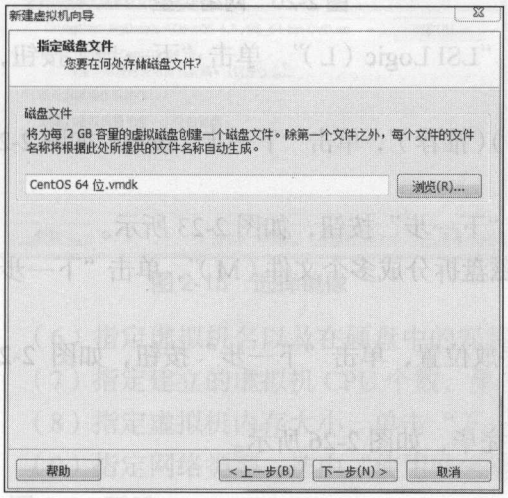



图 2-25 指定磁盘位置

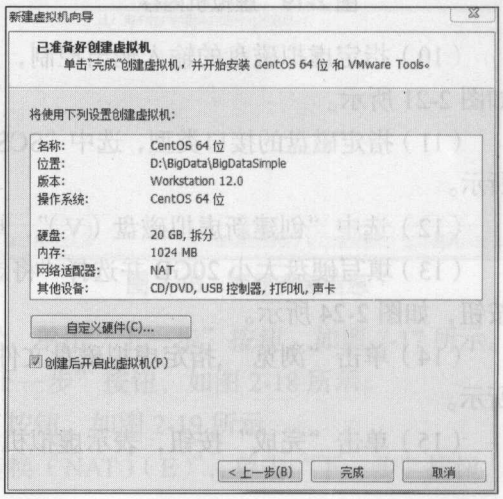



图 2-26 创建完成

（16）进入到桌面，单击“Other”，使用 root 账户登录，如图 2-27 所示。

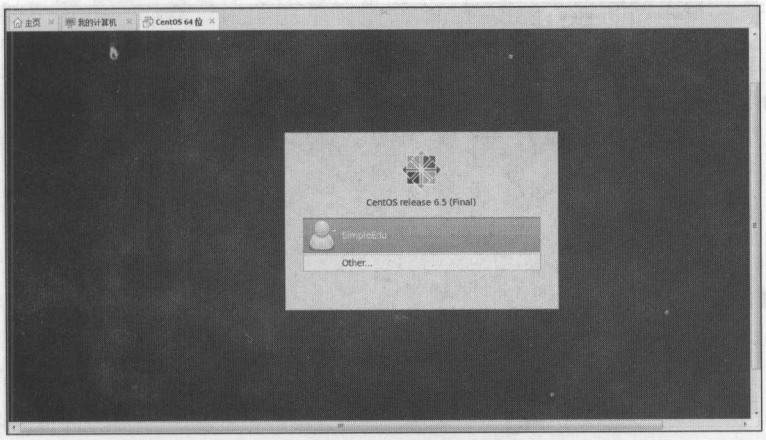



图 2-27　用 root 账户登录

（17）输入用户名为 root，单击“Login”按钮，如图 2-28 所示。

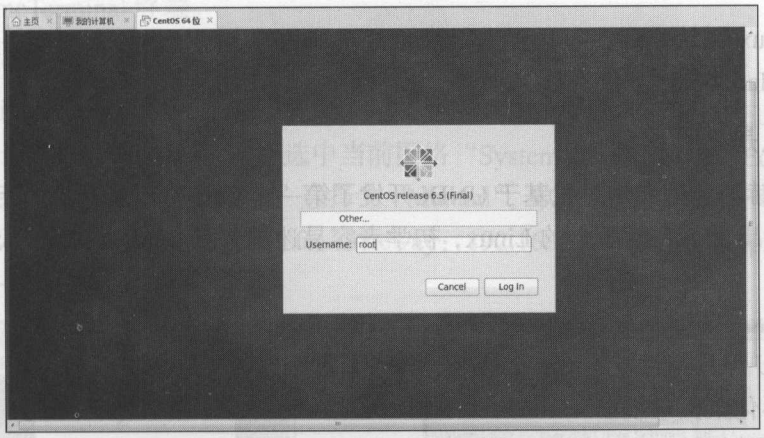



图 2-28　输入用户名

（18）输入密码，密码为 simple，单击“Login”按钮如图 2-29 所示。

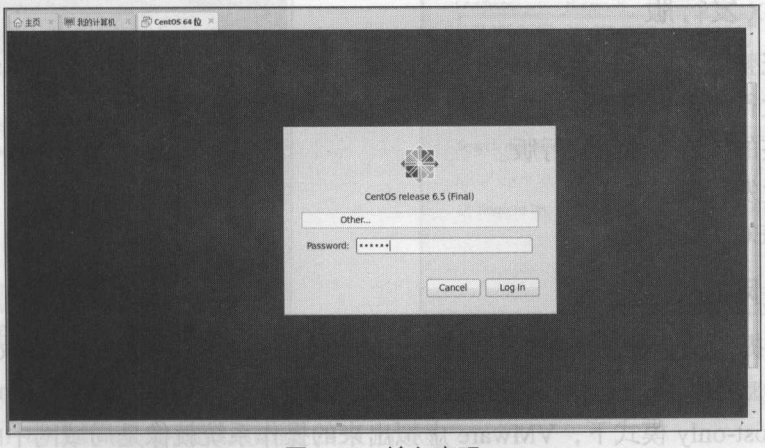



图 2-29　输入密码

（19）登录成功，如图 2-30 所示。

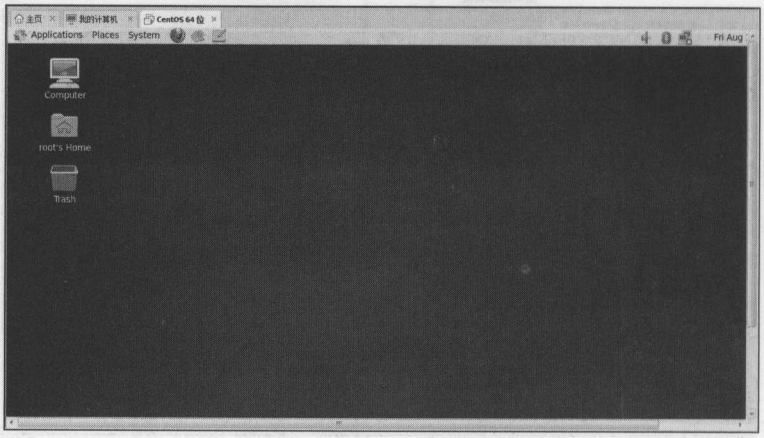



图 2-30　登录成功界面

## 2.2　Linux 配置

安装 Linux 系统之后，有必要进行 Linux 系统的基本配置，只有配置好所需要的系统之后，学习 Hadoop 时才会顺畅。

### 2.2.1　什么是 Linux

Linus 在赫尔辛基大学时代基于 UNIX 开发了第一个版本的 Linux 内核。后来随着 Linux 的快速发展，出现了大量版本的 Linux，初学者容易迷糊，但 Linux 系统内部结构主要由四部分组成：

- Linux 内核。
- GNU 工具组件。
- 图形化桌面环境。
- 应用软件。

Linux 系统的核心是内核，内核控制着计算机系统上的所有硬件和软件。

### 2.2.2　Linux 发行版

通常把完整版本的 Linux 系统称为发行版。一般情况下，不同的发行版针对不同的人群。不同的发行版可以归为三类：

- 完整的核心 Linux 发行版。
- 专业发行版。
- LIVECD 发行版。

### 2.2.3　配置网络

VMware 提供了 3 种工作模式，分别是 bridged 模式、NAT 模式和 host-only 模式。在学习 VMware 虚拟网络时，建议选择 host-only。

（1）在 host-only 模式下，VMware 虚拟出来的操作系统就像是局域网中的一个独立的

主机，可以访问网内的任何一台机器。

（2）在 NAT 模式下，就是让虚拟系统借助 NAT 功能，通过宿主机器所在的网络来访问公网。

（3）在 host-only 模式下，虚拟系统的 TCP/IP 配置信息都可以由 VMnet1 虚拟网络的 DHCP 服务器来动态分配。

### 2.2.4　Linux 终端

配置 Linux 环境一般会使用 Linux 终端，在图形化桌面出现之前，和 Unix 系统交互的唯一方式就是通过 Shell 提供的文本命令行界面（Command Line Interface，CLI）。CLI 只允许输入文本，而且只能显示文本和比较简单的图形。现在 Linux 系统基本上都支持图形化桌面环境，可以实现使用者自由地在 CLI 和图形界面来回切换。

常见的终端有：

- Xterm 终端。
- Konsole 终端。
- GnomeTerminal 终端。

**【案例 2-1】设置静态 IP**

（1）在 Linux 系统可视桌面中，进入 Linux 图形界面，单击右键桌面右上方的两个小电脑图标，单击“Edit connections”，选中当前网络“System eth0”，单击“Edit”按钮，选择“IPv4 Settings”标签页，“Method:”选择为“Manual”，单击“Add”按钮，填写 Address：192.168.0.202，Netmask：255.255.255.0，Gateway：192.168.0.1，单击“Apply”按钮，如图 2-31、图 2-32 所示。

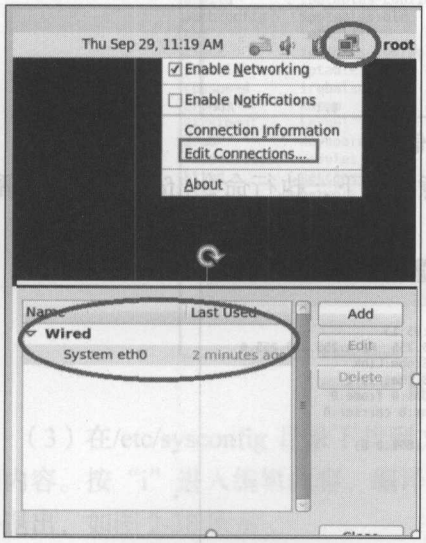



图 2-31　选择网络

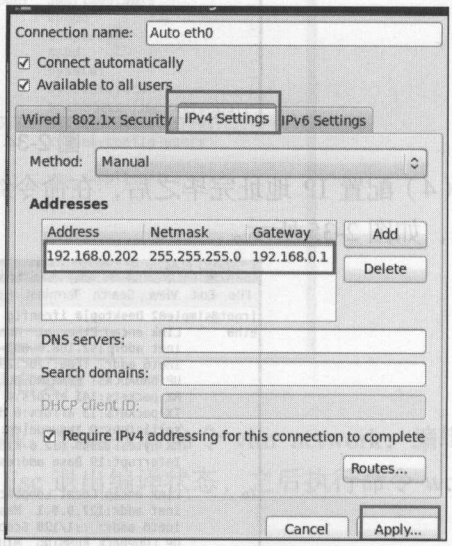



图 2-32　网络信息

（2）在 Linux 系统命令终端，执行命令 cd /etc/sysconfig/network-scripts，切换到该目录并查看该目录下的文件 ifcfg-eth0，如图 2-33 所示。

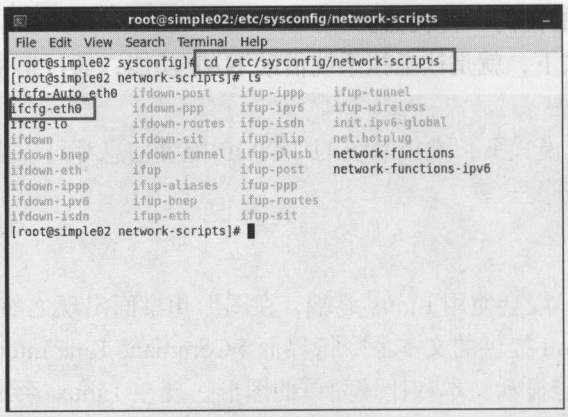



图 2-33　网络配置文件

（3）在 Linux 系统命令终端，执行命令 vim ifcfg-eth0，并修改文件的内容，按“i”键进入编辑内容,编译完成后按 Esc 键退出编译状态,之后执行命令 wq，保存并退出。HWADDR、IPADDR、NETMASK、GATEWAY 的值可以根据自己的本机进行修改，如图 2-34 所示。

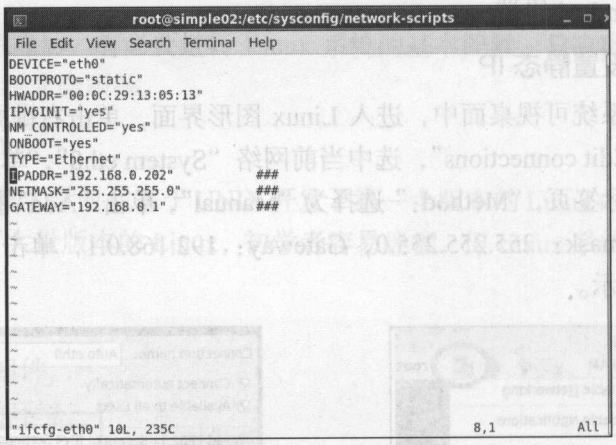



图 2-34　网络配置信息

（4）配置 IP 地址完毕之后，在命令终端的任意目录下，执行命令 ifconfig，查看配置效果，如图 2-35 所示。

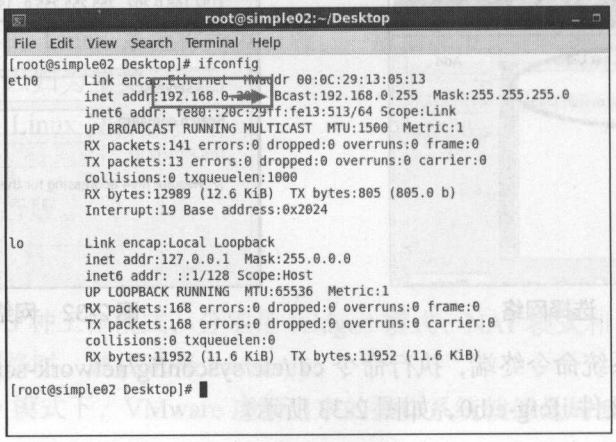



图 2-35　查看

【案例 2-2】修改主机名和映射文件

（1）在 Linux 系统可视桌面中，单击右键鼠标，选择“open in terminal”启动命令终端，弹出命令终端窗口，如图 2-36 所示。

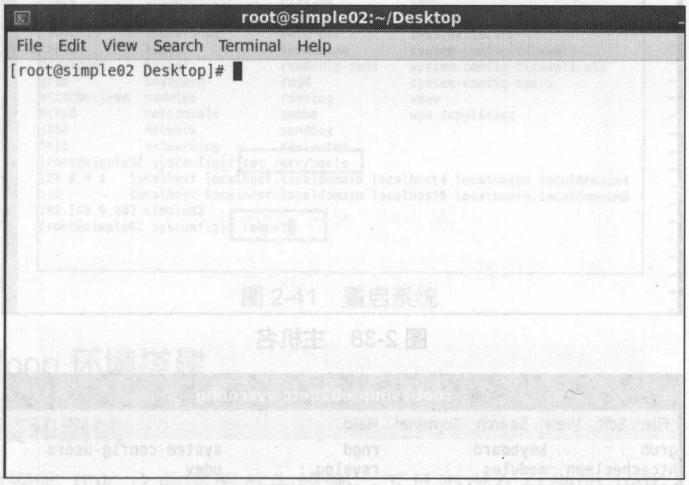



图 2-36 命令终端

（2）启动命令终端，在任何目录下执行命令 cd /etc/sysconfig，切换到该目录并查看目录下的文件，可以发现存在文件 network，如图 2-37 所示。

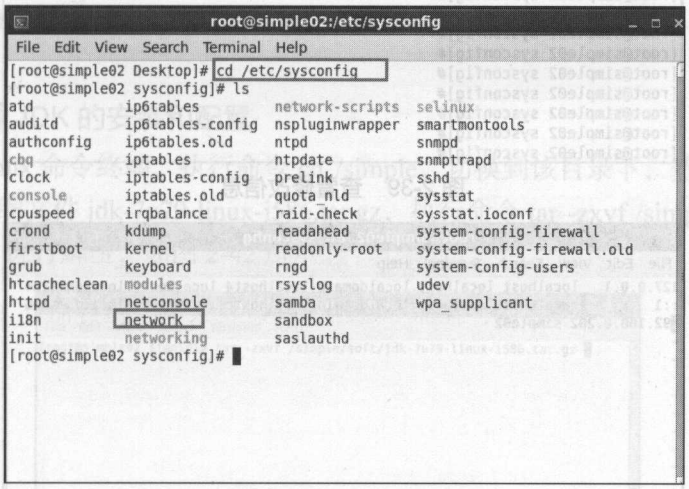



图 2-37 主机名文件

（3）在/etc/sysconfig 目录下找到文件 network，然后执行命令 vim network，编辑其中的内容。按“i”进入编辑内容，编译完成后按 Esc 退出编译状态，之后执行命令 wq 保存并退出，如图 2-38 所示。

（4）在/etc/sysconfig 目录下执行命令 cat network，查看 network 文件中的内容，如图 2-39 所示。

（5）如果要修改主机名和 IP 地址具有映射关系。执行命令 vim /etc/hosts，并编辑其中的内容，按“i”进入编辑内容,编译完成后按 Esc 退出编译状态,之后执行命令 wq 保存并退

出，如图 2-40 所示。

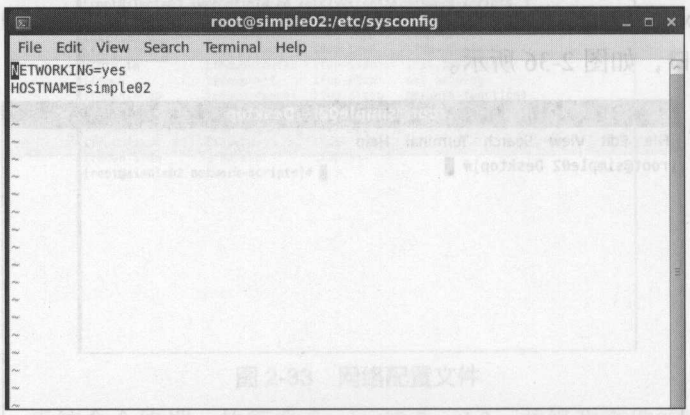



图 2-38　主机名

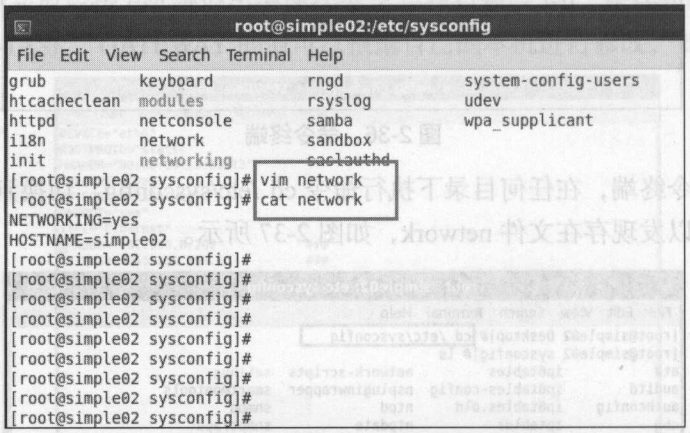



图 2-39　查看修改信息

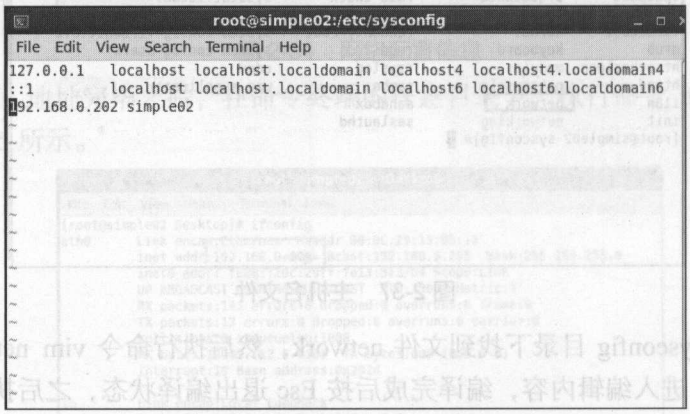



图 2-40　主机名和 ip 映射

（6）当修改完/etc/sysconfig/network 和/etc/hosts 文件之后，需要执行命令 reboot，重新启动系统，主机名才会有效，如图 2-41 所示。

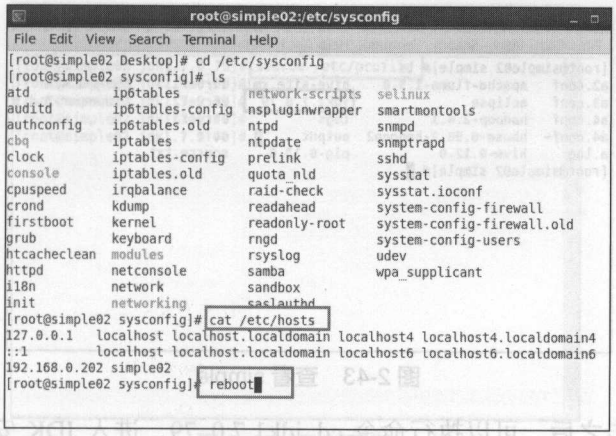



图 2-41 重启系统

## 2.3 Hadoop 环境搭建

### 2.3.1 JDK 安装和测试

Linux 系统中安装 JDK 之前需要首先检测一下是否存在自带的 JDK，如果存在，则需要先进行卸载。检测方法可以通过在命令终端执行命令 java-version。如果发现有存在版本，则需要执行命令 rpm-qa/grepjdk 和 rpm-qa/grep gcj。然后根据查找到的软件，通过执行 yum -y remove 进行卸载。

完成上面的测试步骤之后，可以从网上下载 JDK 的压缩包，通过 tar-zxvf 解压命令实现 JDK 解压。

【案例 2-3】JDK 的安装和配置

（1）启动 Linux 命令终端，执行命令 cd /simple，切换到该目录下，然后对/simple/soft 目录下的 JDK 压缩文件 jdk-7u79-linux-i586.tar.gz，执行命令 tar -zxvf /simple/soft/jdk-7u79-linux-i586.tar.gz，进行解压，如图 2-42 所示。

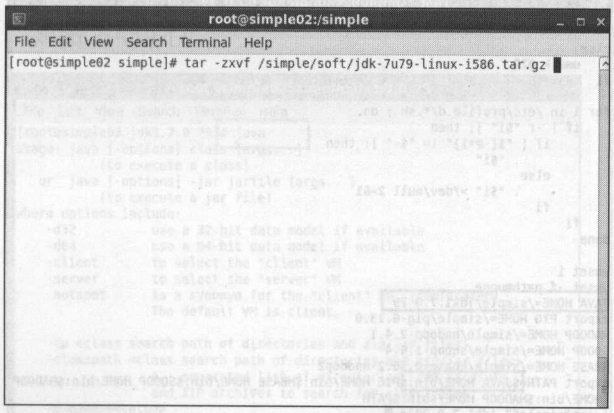



图 2-42 解压文件

（2）解压之后，执行命令 ls /simple，可以看到该目录下多了一个解压后的 JDK 文件，如图 2-43 所示。

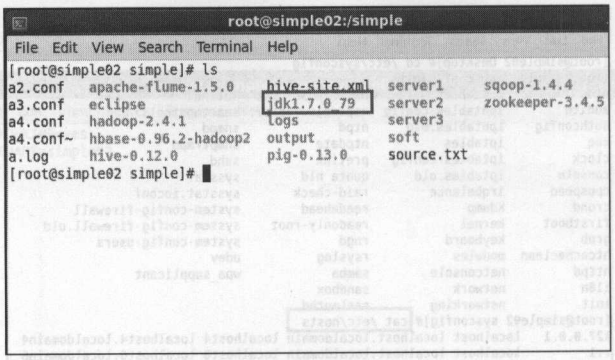



图 2-43　查看 simple

（3）完成上一步之后，可以执行命令 cd jdk1.7.0_79，进入 JDK 安装目录，如图 2-44 所示。

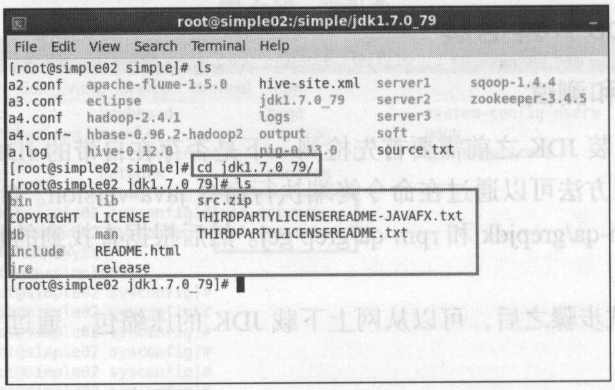



图 2-44　查看 JDK 文件

（4）确定解压无误之后，此时需要配置 JDK 环境变量，执行命令 vim /etc/profile，并按“i”进入编辑内容，编译完成后按 Esc 退出编译状态，之后执行命令 wq 保存并退出，如图 2-45 所示。

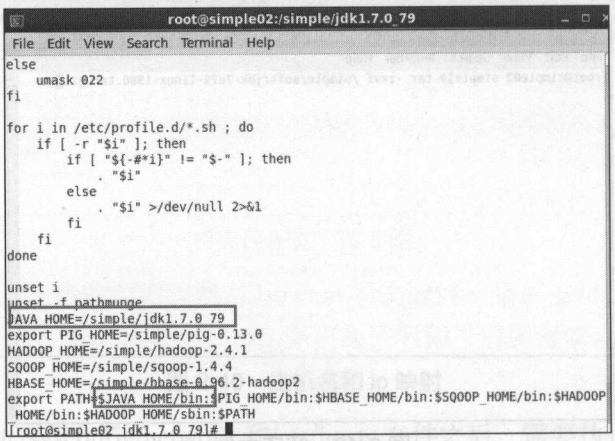



图 2-45　环境变量

（5）编辑完后进行配置文件刷新，执行命令 source /etc/profile，刷新配置，配置文件中的信息才会生效，如图 2-46 所示。

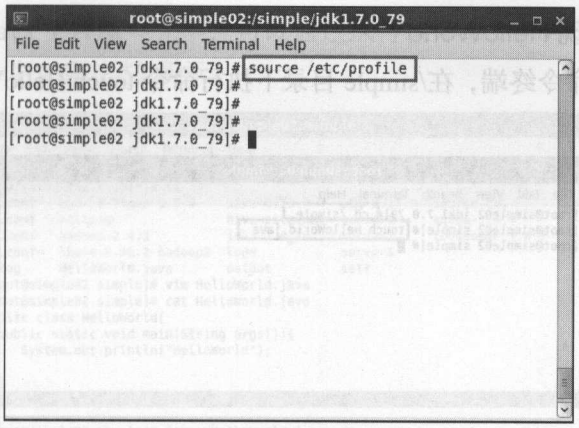



图 2-46 刷新配置

（6）完成以上步骤之后，需要测试环境变量是否配置成功，只需要在任何目录下执行 javac 命令，如果提示“找不到命令”，则表示配置未成功，如图 2-47 所示。

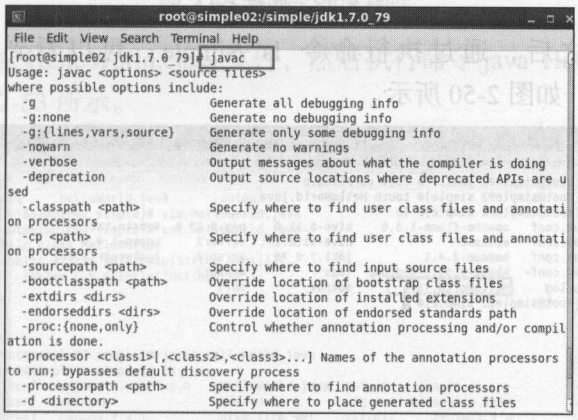



图 2-47 配置成功 1

（7）完成以上步骤之后执行 java 命令，如果提示“找不到命令”，则表示配置未成功，否则，表示配置成功，如图 2-48 所示。

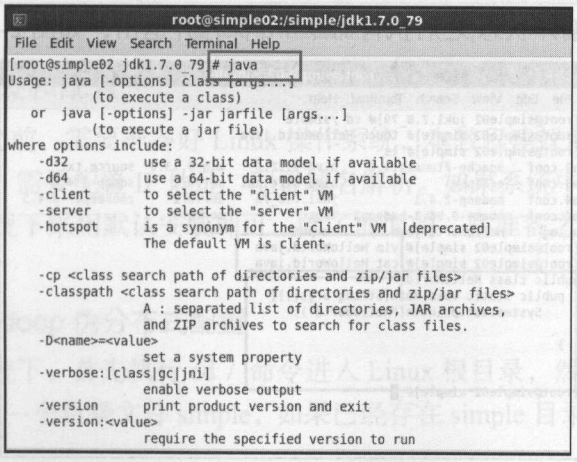



图 2-48 配置成功 2

【案例 2-4】实现 HelloWorld

（1）启动 Linux 命令终端，在/simple 目录下执行命令 touch HelloWorld.java，创建一个文件，如图 2-49 所示。

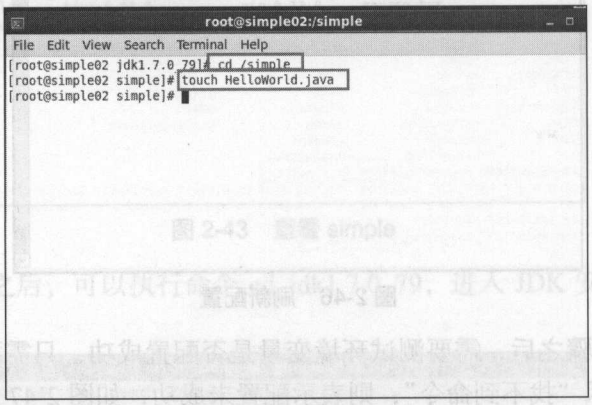



图 2-49　新建文件

（2）创建文件之后，通过执行命令 ls /simple，可以看到该目录下多了一个 HelloWorld.java 文件，如图 2-50 所示。

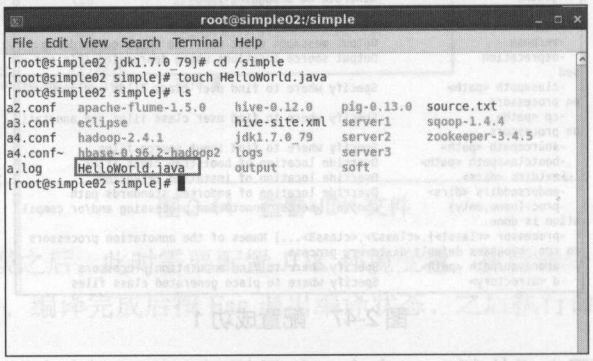



图 2-50　查看文件

（3）完成上一步之后，可以执行命令 vim HelloWorld.java，并按“i”进入编辑内容,编译完成后按 Esc 退出编译状态,之后执行命令 wq 保存并退出，如图 2-51 所示。

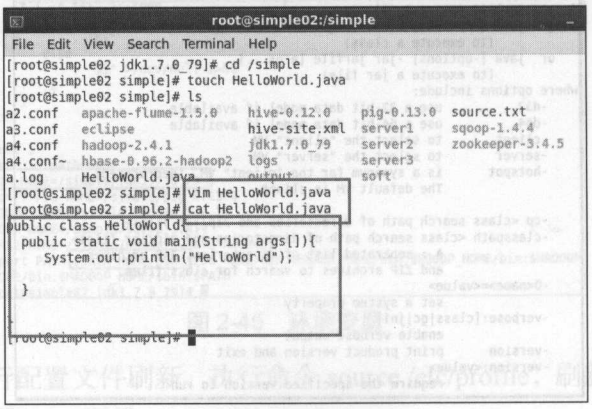



图 2-51　编译文件

（4）HelloWorld.java 文件编写完毕，可以通过执行编译命令 javac HelloWorld.java，生成 HelloWorld.class 文件,如图 2-52 所示。

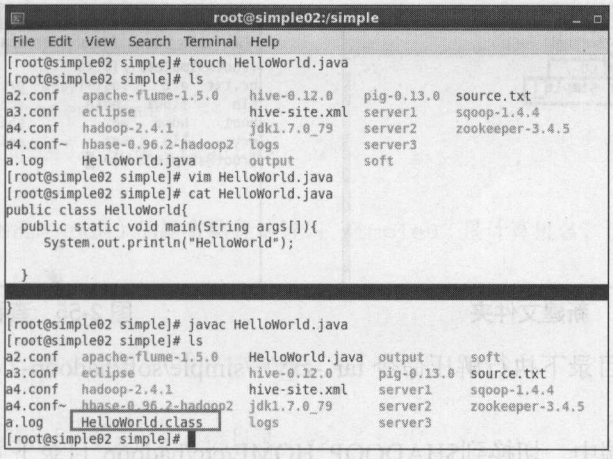



图 2-52 运行.java 文件

（5）编译之后生成 HelloWorld.class 文件，然后执行命令 java HelloWorld,执行 class 文件并输出结果，如图 2-53 所示。

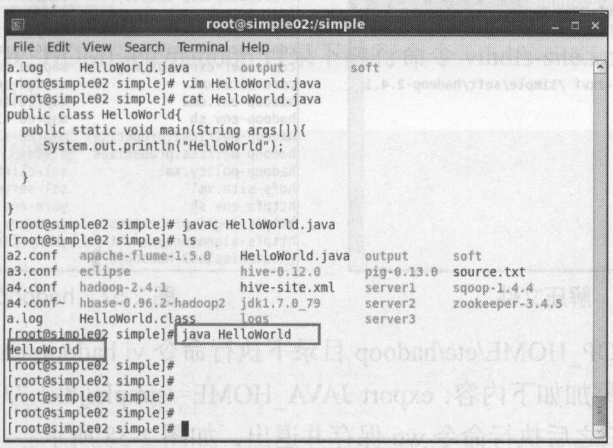



图 2-53 查看效果

## 2.3.2 Hadoop 安装和配置

在安装 Hadoop 之前，需要准备好 Linux 操作系统，本教程建议使用 Centos 64 位操作系统，安装系统之后，需要配置 IP 地址，配置域名解析。如果系统中没有 WGET 和 SSH，需要先安装，一般情况下采用默认安装即可。除了前面的一些准备之外，JDK 环境的搭建是必不可少的。

**【案例 2-5】Hadoop 伪分布式配置**

（1）在 Linux 系统下，首先执行 cd / 命令进入 Linux 根目录，然后在该目录下执行命令 mkidr simple，创建一个目录文件 simple，如果已经存在 simple 目录，不需要再创建，如图 2-54 所示。

（2）创建 simple 目录之后，在 Linux 根目录下通过执行命令 ls，验证是否创建成功，如图 2-55 所示。

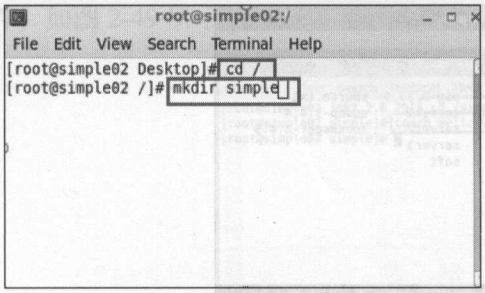



图 2-54　新建文件夹

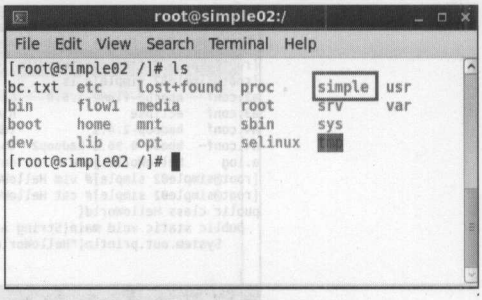



图 2-55　查看文件夹

（3）在 simple 目录下执行解压命令 tar -zxvf /simple/soft/hadoop-2.4.1.tar.gz，解压压缩包，如图 2-56 所示。

（4）在命令终端中，切换到$HADOOP_HOME/etc/hadoop 目录下并查看该目录下的文件列表，如图 2-57 所示。

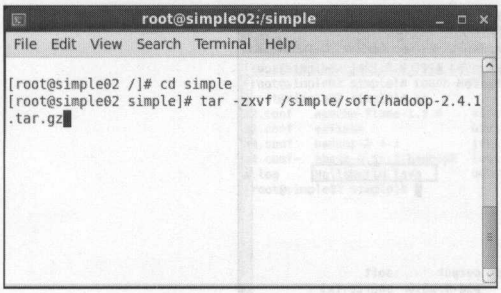



图 2-56　解压文件

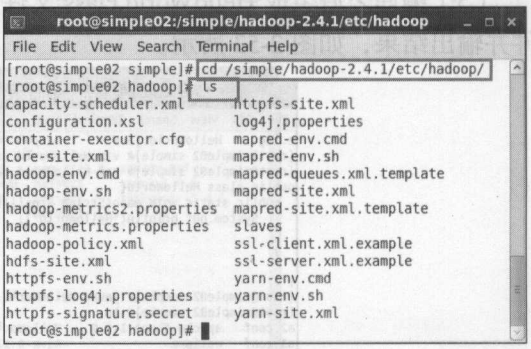



图 2-57　hadoop 配置文件

（5）在$HADOOP_HOME/etc/hadoop 目录下执行命令 vi hadoop-env.sh，按“i”键进入编辑内容,在文件中添加如下内容: export JAVA_HOME=/simple/jdk1.7.0_79，编译完成后按 Esc 键退出编译状态,之后执行命令 wq 保存并退出，如图 2-58 所示。

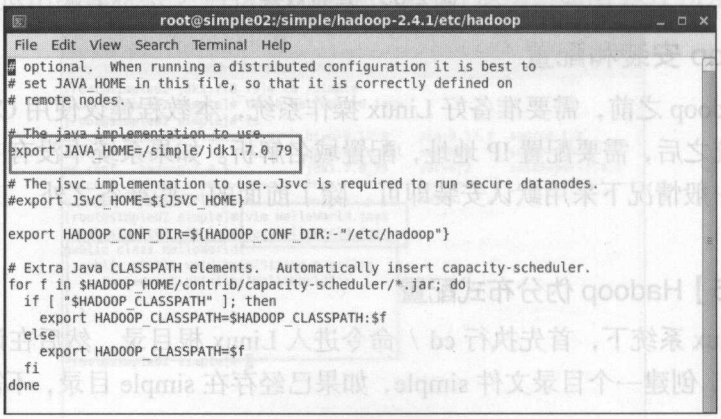



图 2-58　修改配置文件

（6）在$HADOOP_HOME/etc/hadoop 目录下执行命令 vi core-site.xml，并修改配置文件 core-site.xml，内容如下(实际修改不需要写中文注释)。

```
<!--指定 fs 的缺省名称-->
<property>
<name>fs.default.name</name>
<value>hdfs://simple02:9000</value>
</property>
<!--指定 HDFS 的（NameNode）的缺省路径地址 ： simple02 是计算机名，也可以是 ip 地址-->
<property>
<name>fs.defaultFS</name>
  <value>hdfs://simple02:9000</value>
</property>
<!-- 指定 hadoop 运行时产生文件的存储目录 -->
<property>
  <name>hadoop.tmp.dir</name>
  <value>/simple/hadoop-2.4.1/tmp</value>
</property>
```

（7）在$HADOOP_HOME/etc/hadoop 目录下执行命令 vihdfs-site.xml，并修改配置文件 hdfs-site.xml，内容如下。

```
<!-- 指定 HDFS 副本的数量 -->
<property>
  <name>dfs.replication</name>
  <value>1</value>
  </property>
  <property>
      <name>dfs.name.dir</name>
      <value>/simple/Hadoop-2.4.1/hdfs/name</value>
</property>
<property>
      <name>dfs.data.dir</name>
     <value>/simple/hadoop-2.4.1/hdfs/data</value>
</property>
```

（8）在$HADOOP_HOME/etc/hadoop 目录下查看是否有配置文件 mapred-site.xml。目录下默认情况下没有该文件，可通过执行命令 mv mapred-site.xml.template mapred-site.xml，修改一个文件的命名，然后执行编辑文件命令 vi mapred-site.xml 并修改该文件内容。

```
<!-- 指定 mr 运行在 yarn 上 -->
<property>
```

```
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

（9）在$HADOOP_HOME/etc/hadoop 目录下执行命令 vi yarn-site.xml，并修改配置文件内容如下。

```
<!-- 指定 YARN 的老大（ResourceManager）的地址 -->
<property>
    <name>yarn.resourcemanager.hostname</name>
    <value>simple02</value>
</property>
<!-- reducer 获取数据的方式 -->
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
```

（10）执行命令 vi /etc/profile，把 Hadoop 的安装目录配置到环境变量中，如图 2-59 所示。

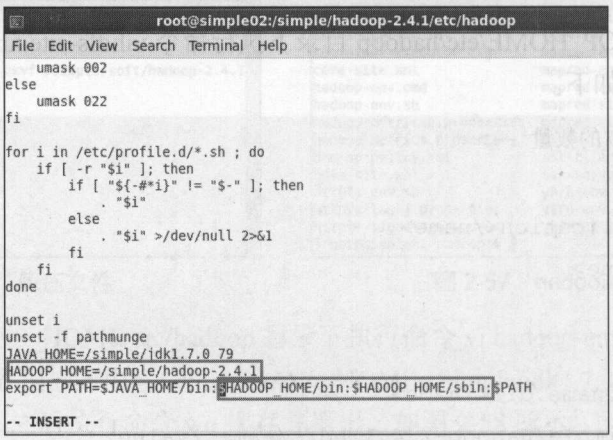



图 2-59　环境变量

（11）然后让配置文件生效，执行命令 source /etc/profile，如图 2-60 所示。

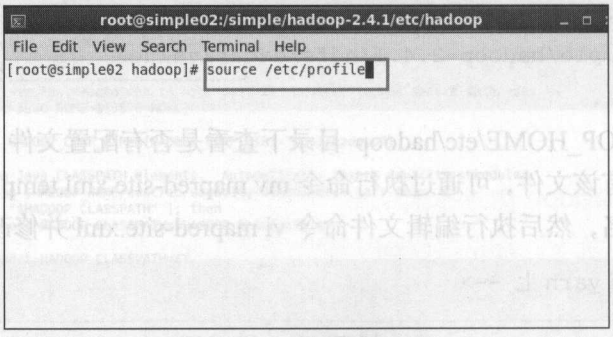



图 2-60　刷新配置文件

（12）格式化 NameNode。在任意目录下（配置 Hadoop 环境变量的情况下）执行命令 hdfs namenode -format 或者 hadoop namenode -format，实现格式化， 如图 2-61 所示。

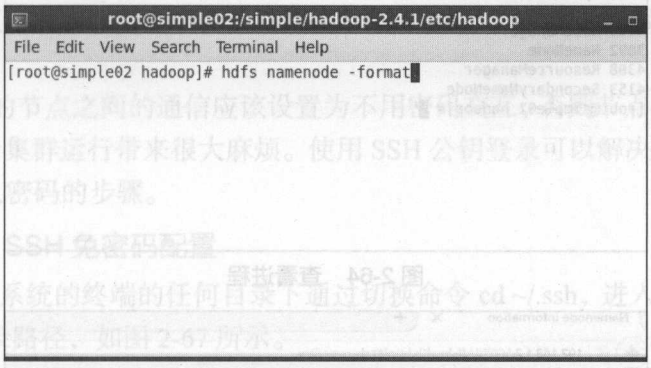



图 2-61　namenode 格式化

（13）启动 Hadoop 进程，首先执行命令 start-dfs.sh，启动 HDFS 系统，如图 2-62 所示。

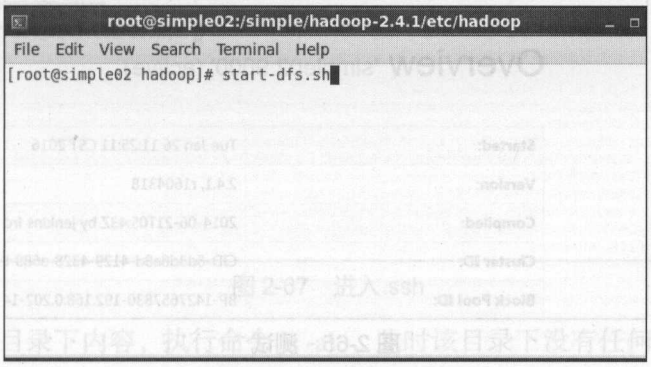



图 2-62　启动 HDFS

（14）启动 Yarn，执行命令 start-yarn.sh，启动 Yarn 计算进程，如图 2-63 所示。

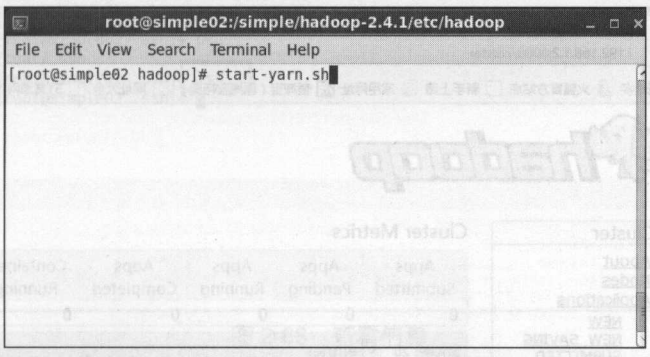



图 2-63　启动 Yarn

（15）启动之后，在任意目录下执行 jps 命令验证进程是否正常启动，如图 2-64 所示。

（16）测试 HDFS 和 Yarn（推荐火狐浏览器），首先在浏览器地址栏中输入 http://192.168.1.2:50070（HDFS 管理界面）(本 IP 为自己虚拟机上面的 IP，端口不变)，如图 2-65 所示。

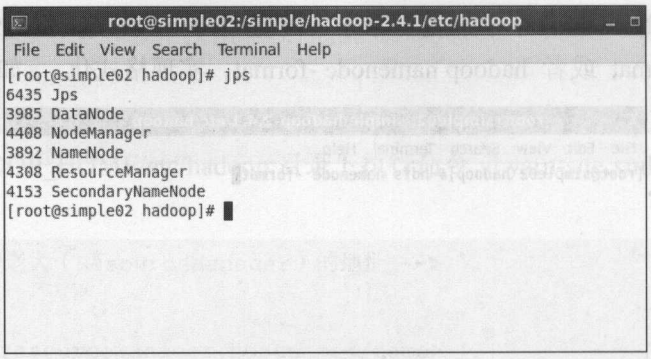



图 2-64　查看进程

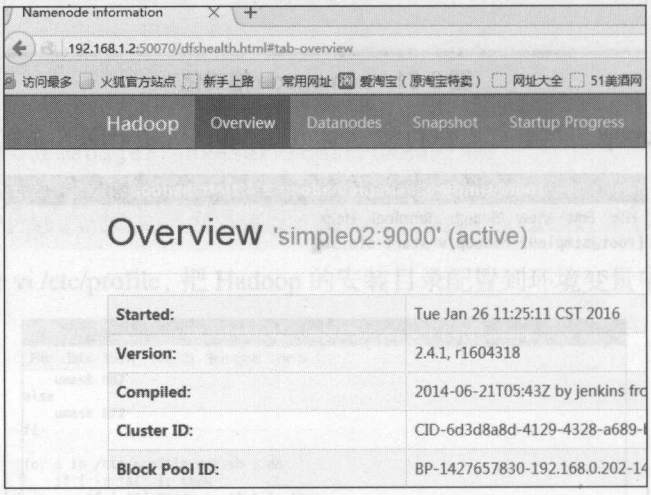



图 2-65　测试

（17）在浏览器的地址栏中输入 http://192.168.1.2:8088 （MR 管理界面）（本 IP 为自己虚拟机上面的 IP，端口不变），如图 2-66 所示。

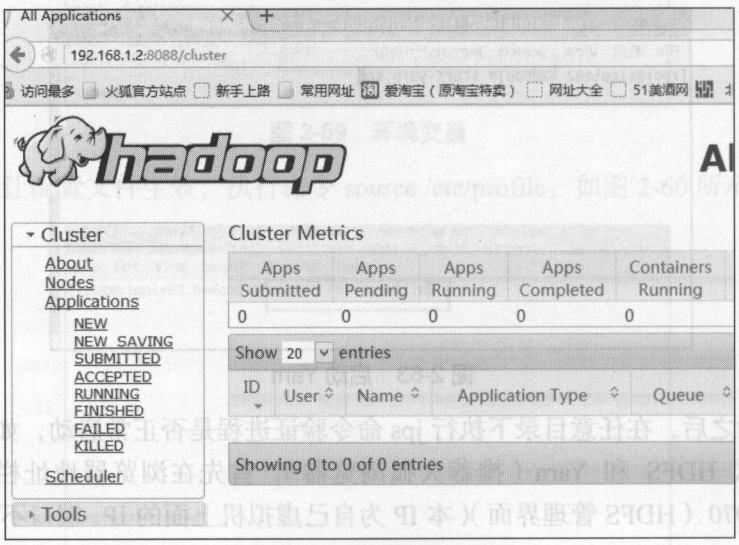



图 2-66　测试 Hadoop

### 2.3.3 SSH 免密码配置

SSH 是 Secure Shell 的缩写，由 IETF 的网络工作小组制定。SSH 是建立在应用层和传输层上的安全协议，专为远程登录会话和其他网络服务提供安全性的协议。目前 SSH 相对比较可靠。

Hadoop 集群的节点之间的通信应该设置为不用密码交互，否则节点之间每次通信都需要输入密码，会给集群运行带来很大麻烦。使用 SSH 公钥登录可以解决这个问题，省略掉节点通信需要输入密码的步骤。

【案例 2-6】SSH 免密码配置

（1）在 Linux 系统的终端的任何目录下通过切换命令 cd ~/.ssh，进入到.ssh 目录下并通过 pwd 查看该目录路径，如图 2-67 所示。

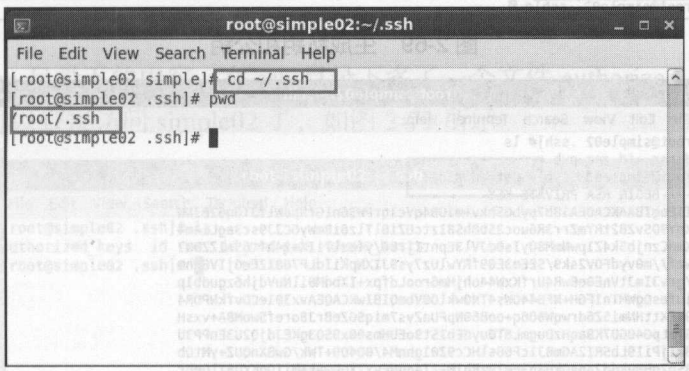



图 2-67 进入.ssh

（2）查看.ssh 目录下内容，执行命令 ls -al ，此时该目录下没有任何文件和文件夹，如图 2-68 所示。

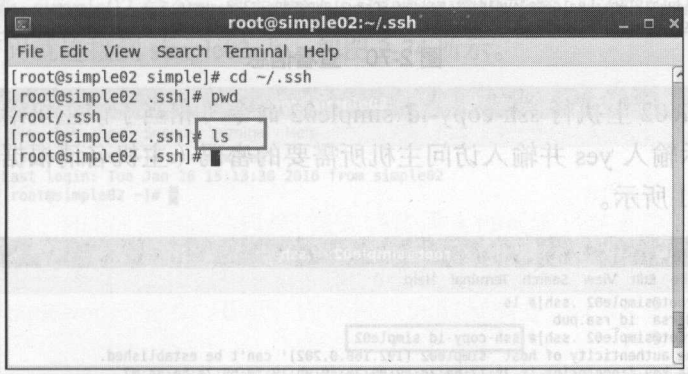



图 2-68 查看信息

（3）在 Linux 系统命令框的.ssh 目录下执行命令 ssh-keygen -t rsa （连续按四次回车），执行完上面命令后，会生成两个文件 id_rsa（私钥）、id_rsa.pub（公钥），如图 2-69 所示。

（4）查看公钥和私钥内容，执行命令 cat id_rsa 和 cat id_rsa.pub ，查看公钥和私钥的信息，如图 2-70 所示。

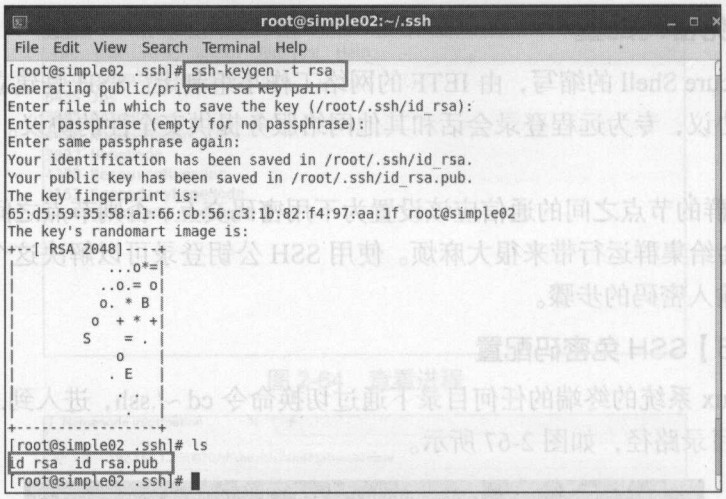



图 2-69　生成私钥和公钥

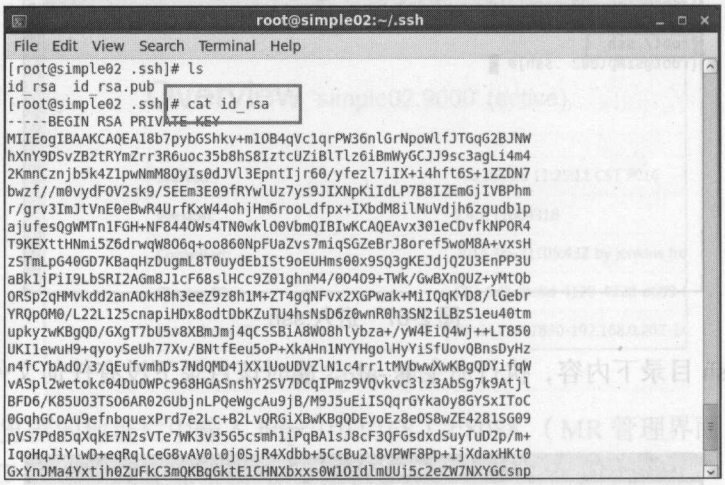



图 2-70　查看信息

（5）在 simple02 上执行 ssh-copy-id simple02 命令（相当于该主机给自身设置免密码登录），根据提示输入 yes 并输入访问主机所需要的密码（主机名请根据实际虚拟机的主机名），如图 2-71 所示。

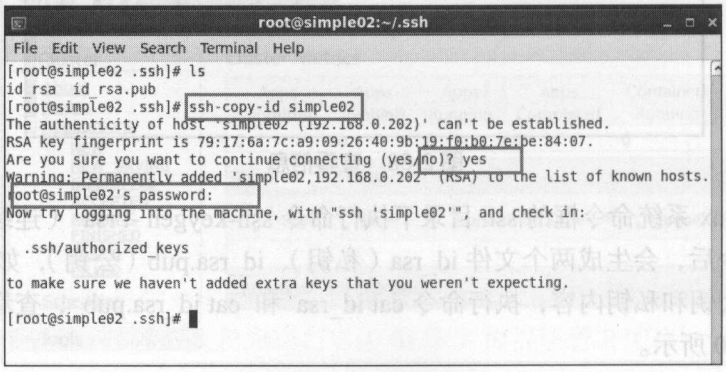



图 2-71　拷贝至本机

（6）在 simple02 机器上切换到.ssh 目录 cd ~/.ssh ，如图 2-72 所示。

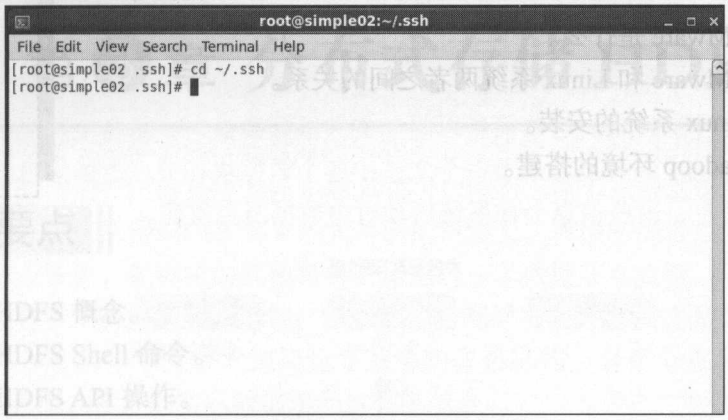



图 2-72　切换.ssh 目录

（7）查看.ssh 目录下的内容，在.ssh 目录下多了一个文件 authorized_key，其内容就是密码值,此时就可以直接访问 simple02 了，如图 2-73 所示。

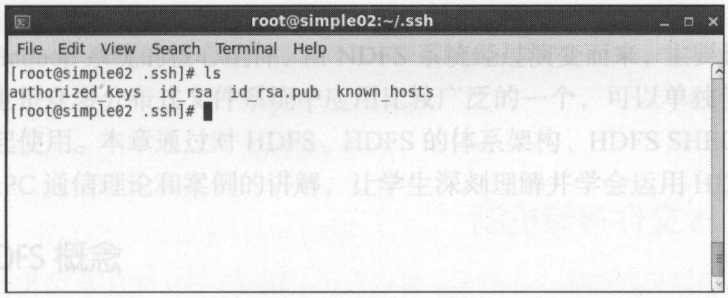



图 2-73　查看目录内容

（8）执行 ssh simple02 命令，然后输入一次访问密码之后，以后再访问 simple02 主机即可不用输入密码就连接到 simple02 上，如图 2-74 所示。

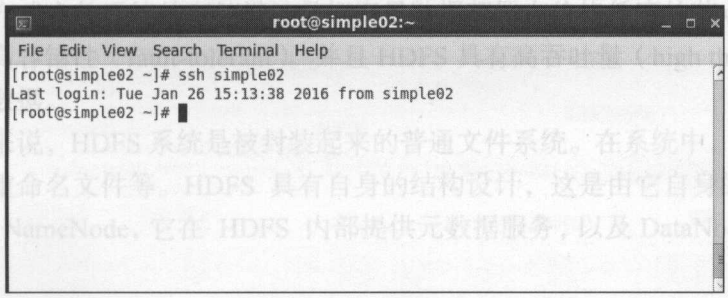



图 2-74　连接本机

## 本章小结

本章是对 Hadoop 环境的搭建进行介绍，首先是介绍了一下 Linux 环境和 Linux 配置，然后介绍 Hadoop 基础环境的搭建，例如 JDK 的安装与配置，Hadoop 伪分布式环境的搭建以及 SSH 的介绍和免密码配置。

## 习题

1. 简述 VMware 是什么。
2. 简述 VMware 和 Linux 系统两者之间的关系。
3. 学会 Linux 系统的安装。
4. 学会 Hadoop 环境的搭建。

扫一扫在线测

# 第3章 分布式存储 HDFS

## 本章要点

- 了解 HDFS 概念。
- 熟练 HDFS Shell 命令。
- 掌握 HDFS API 操作。
- 理解 RPC 通信。

## 引　言

HDFS 是 Hadoop 系统的核心组件，由 NDFS 系统经过演变而来，主要解决海量大数据存储的问题，也是众多分布式文件系统中应用比较广泛的一个，可以单独使用，一般配合 MapReduce 一起使用。本章通过对 HDFS、HDFS 的体系架构、HDFS SHELL 命令、HDFS API 的操作、RPC 通信理论和案例的讲解，让学生深刻理解并学会运用 HDFS 系统。

## 3.1 HDFS 概念

### 3.1.1 HDFS 简介

Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）是 Hadoop 核心组件之一。支持以流式数据访问模式来存取超大文件，活动在集群之上。

HDFS 分布式文件系统的存储设计是把海量数据部署在价格低廉的节点上，通过这种方式可以解决高容错性（fault-tolerant）。并且 HDFS 具有高吞吐量（high throughput）来访问应用程序的数据。

对于用户来说，HDFS 系统是被封装起来的普通文件系统。在系统中，用户可以创建、删除、移动或重命名文件等。HDFS 具有自身的结构设计，这是由它自身的特点决定的。这些节点包括 NameNode，它在 HDFS 内部提供元数据服务，以及 DataNode，它为 HDFS 提供存储块。

### 3.1.2 HDFS 设计思路和理念

HDFS 设计的时候首要是针对超大文件存储的，对于小的文件访问和存储速度反而会降低。另外 HDFS 采用了高效的流式访问模式，明显的特点就是“一次写入，多次读取”；再有就是它运行在普通的硬件之上，即使硬件故障，也可通过容错来保证数据的高可用。

## 3.2 HDFS 体系结构

HDFS 的体系结构如图 3-1 所示。HDFS 集群有两类节点并以管理者—工作者模式运行，即一个管理者和多个工作者。NameNode 管理文件系统的命名空间。它维护着文件系统树及其中的所有文件和目录。这些信息以两个文件保存磁盘中：命名空间镜像文件和编辑日志文件。NameNode 同时也记录着每个文件中各个数据块在节点上的信息，但它并不永久保存块的位置信息，这些信息会在系统启动时由数据节点重建。

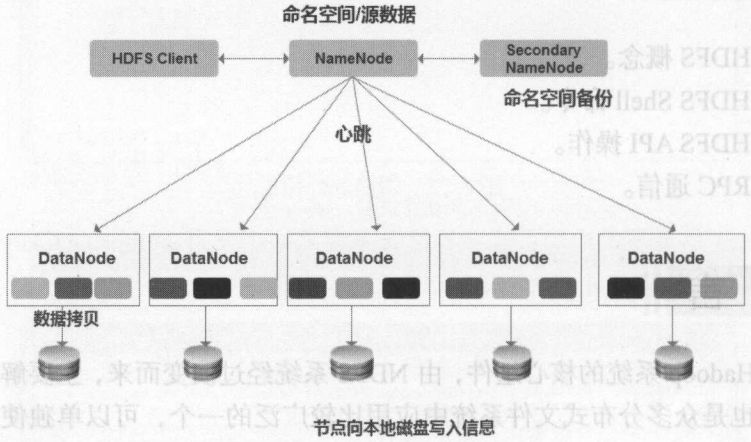



图 3-1　HDFS 体系结构

## 3.3 HDFS 文件存储机制

HDFS 文件存储机制如图 3-2 所示。可以看出，HDFS 是一个主从结构，一个 HDFS 集群有一个名字节点，它是一个管理文件命名空间和调节客户端访问文件的主服务器，当然还有一些数据节点，通常是一个节点一个机器，它来管理对应节点的存储。HDFS 对外开放文件命名空间并允许用户数据以文件形式存储。

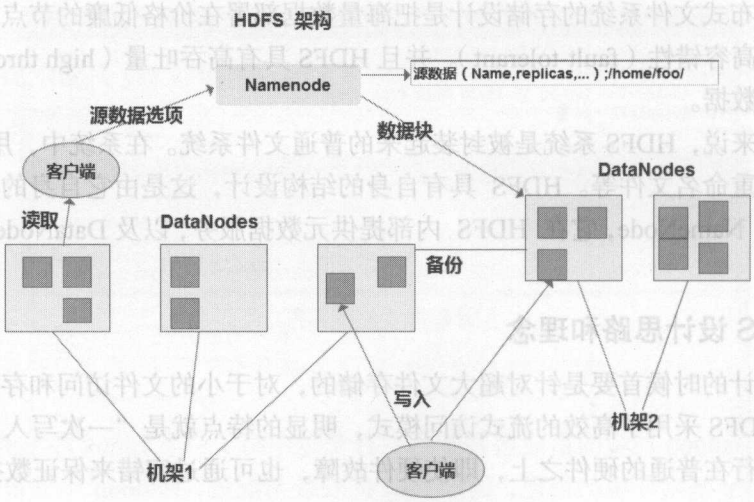



图 3-2　HDFS 文件存储机制

HDFS 的内部机制是将一个文件分割成一个或多个块，这些块被存储在一组数据节点中。名字节点用来操作文件命名空间的文件或目录操作，如打开、关闭、重命名等。它同时确定块与数据节点的映射。数据节点负责来自文件系统客户的读写请求。数据节点同时还要执行块的创建、删除和来自名字节点的块复制指令。

名字节点和数据节点都是运行在普通的机器之上的软件，机器典型的都是 GNU/Linux，HDFS 是用 Java 编写的，任何支持 Java 的机器都可以运行名字节点或数据节点，利用 Java 语言的超轻便性，很容易将 HDFS 部署到大范围的机器上。典型的部署是由一个专门的机器来运行名字节点软件，集群中的其他每台机器运行一个数据节点实例。体系结构不排斥在一个机器上运行多个数据节点的实例，但是实际的部署不会有这种情况。

集群中只有一个名字节点极大地简化了系统的体系结构。名字节点是仲裁者和所有 HDFS 元数据的仓库，用户的实际数据不经过名字节点。

**【案例 3-1】验证 DataNode 存储的块信息**

（1）查看服务状态。在终端命令框中，执行命令 jps，来查看 Hadoop 进程执行的状态，并没有启动 Hadoop 进程（查看 Hadoop 进程是否已经启动，如未启动，启动服务），如图 3-3 所示。

```
[root@host01 Desktop]# jps
2685 Jps
[root@host01 Desktop]#
```

图 3-3 进程状态

（2）启动 Hadoop 进程。安装和配置 Hadoop 之后，需要启动 Hadoop 进程，执行命令 start-all.sh，可以把 HDFS 和 Yarn 的服务全部启动起来（一般不建议使用该命令，建议使用 start-dfs.sh 和 start-yarn.sh 分别启动），如图 3-4 所示。

```
[root@host01 .ssh]# jps
3793 Jps
[root@host01 .ssh]# start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [host01]
host01: starting namenode, logging to /simple/hadoop-2.4.1/logs/hadoop-root-name
node-host01.out
localhost: starting datanode, logging to /simple/hadoop-2.4.1/logs/hadoop-root-d
atanode-host01.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /simple/hadoop-2.4.1/logs/hadoop
-root-secondarynamenode-host01.out
starting yarn daemons
starting resourcemanager, logging to /simple/hadoop-2.4.1/logs/yarn-root-resourc
emanager-host01.out
localhost: starting nodemanager, logging to /simple/hadoop-2.4.1/logs/yarn-root-
nodemanager-host01.out
[root@host01 .ssh]# jps
4006 DataNode
3913 NameNode
4418 NodeManager
4324 ResourceManager
4910 Jps
4174 SecondaryNameNode
[root@host01 .ssh]#
```

图 3-4 启动 Hadoop

（3）在命令终端执行命令 cd /simple/hadoop-2.4.1/etc/hadoop，进入 Hadoop 配置文件目录并执行命令 cat hdfs-site.xml，查看配置文件 hdfs-site.xml 的内容，Hadoop 数据副本数量有属性为 dfs.replication，如图 3-5 所示。

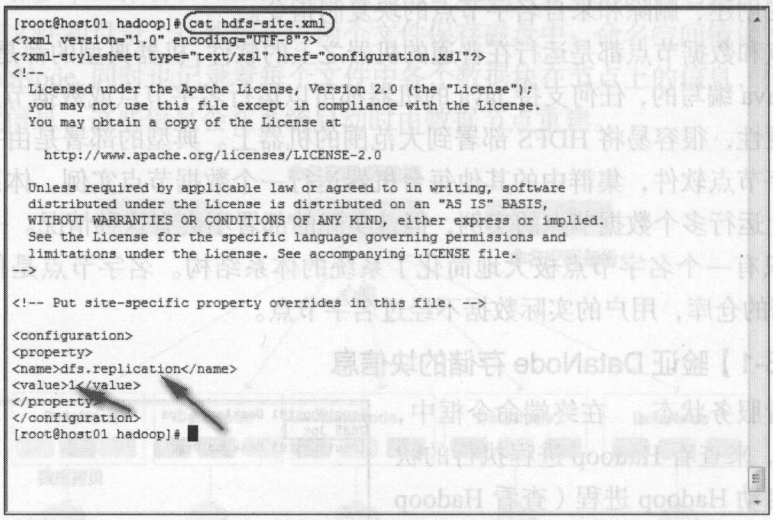



图 3-5　查看配置文件

（4）将目录“/soft”下的 jdk-7u79-linux-x64.tar.gz 上传到 HDFS 的根目录下，执行命令 hadoop fs　- put /soft/ jdk-7u79-linux-x64.tar.gz /，完成文件上传，然后执行命令 hadoop fs -ls /，查看是否上传成功，如图 3-6 所示。

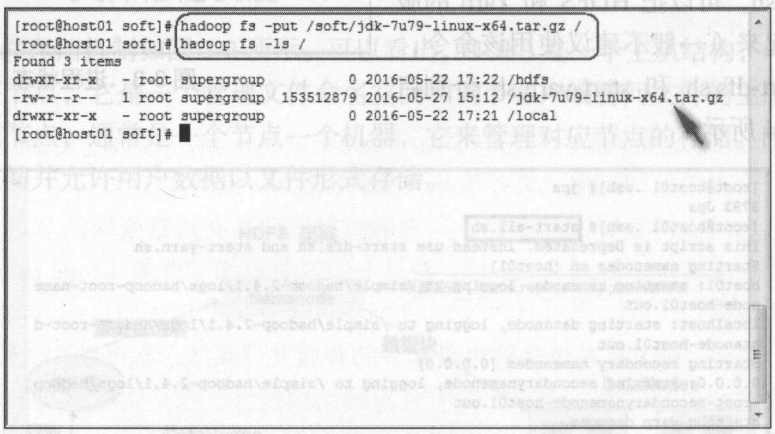



图 3-6　上传文件

（5）进入存放 DataNode 的 Block 目录，执行命令 cd /simple/hadoop-2.4.1/tmp/dfs/data/current/BP*/current/finalized 进入到指定目录，然后执行命令 ls　- lrt，列出 Block 列表信息，如图 3-7 所示。

（6）最大 Block 大小=134217728byte/1024/1024=128MB，总文件大小：134217728+19295151=153512879（byte）。

与源文件大小比较，可以得出结论：hadoop dfs 根据默认值将文件拆分成最大为 128MB（默认）大小 Block 数据块，如图 3-8 所示。

```
[root@host01 /]# cd /simple/hadoop-2.4.1/tmp/dfs/data/current/BP*/current/finalized
[root@host01 finalized]# pwd
/simple/hadoop-2.4.1/tmp/dfs/data/current/BP-1630232954-192.168.0.131-1463830082865/cur
rent/finalized
[root@host01 finalized]# ls -lrt
total 151124
-rw-r--r--. 1 root root       39 May 22 17:37 blk_1073741826_1002.meta
-rw-r--r--. 1 root root     3589 May 22 17:37 blk_1073741826
-rw-r--r--. 1 root root       15 May 22 17:38 blk_1073741827_1003.meta
-rw-r--r--. 1 root root      843 May 22 17:38 blk_1073741827
-rw-r--r--. 1 root root     3589 May 22 17:38 blk_1073741828
-rw-r--r--. 1 root root       39 May 22 17:38 blk_1073741828_1004.meta
-rw-r--r--. 1 root root       15 May 22 17:38 blk_1073741829_1005.meta
-rw-r--r--. 1 root root      843 May 22 17:38 blk_1073741829
-rw-r--r--. 1 root root  1048583 May 27 15:12 blk_1073741841_1017.meta
-rw-r--r--. 1 root root 134217728 May 27 15:12 blk_1073741841
-rw-r--r--. 1 root root    15751 May 27 15:12 blk_1073741842_1018.meta
-rw-r--r--. 1 root root 19295151 May 27 15:12 blk_1073741842
[root@host01 finalized]#
```

图 3-7 列表信息

```
[root@host01 soft]# ls -lsr /soft
total 271236
149920 -rw-r--r--. 1 root root 153512879 Jun  2  2015 jdk-7u79-linux-x64.tar.gz
121316 -rw-r--r--. 1 root root 124227772 Jul  9  2015 hadoop-2.4.1.tar_64位.gz
[root@host01 soft]#
```

图 3-8 查看文件

## 3.4 HDFS Shell 介绍

和 HDFS 系统交互可以使用 HDFS Shell 命令，这也是最直接而且比较简单的和 HDFS 交互的一种方式。调用文件系统(FS)Shell 命令可以使用 bin/hadoop fs 的形式。

所有的 FS Shell 命令使用 URI 路径作为参数。

URI 格式是 scheme://authority/path。URI 的 scheme 是 hdfs，表示分布式 HDFS 系统，scheme 是 file，表示是本地系统。其中 scheme 和 authority 参数都是可选的，如果未加指定，就会使用配置中指定的默认 scheme。

例如/parent/child 可以表示成 hdfs://namenode:namenodePort/parent/child，或者更简单的/parent/child（假设配置文件是 namenode:namenodePort）。

### 3.4.1 命令格式

文件系统 shell 包含各种 shell like 命令，可以直接和 HDFS 文件系统进行交互，就像对其他系统的支持一样，例如 Local FS, HFTP FS, S3 FS。shell 命令执行格式如下：

```
hadoop fs {args}
hadoop dfs {args}
hdfs dfs {args}
```

说明如下。

hadoop fs：使用面最广，可以操作任何文件系统。

hadoop dfs 与 hdfs dfs：只能操作 HDFS 相关文件系统（包括与 Local FS 间的操作），前者已经 Deprecated，一般使用后者。

### 3.4.2 HDFS 用户命令

用户命令对于集群的用户是很有用的，命令也比较多，例如创建文件、创建文件夹、移动文件、删除文件、重命名、合并文件等，命令的使用方法和格式与 Linux 的命令有很大相似之处，对于用户记忆命令是个难题，但是用户可以借助系统提供的帮助命令查看命令和命令参数，这给用户带来极大的方便，只需要执行命令 hdfs dfs -help，查看命令列表，如图 3-9 所示。

```
[root@simple01 ~]# hdfs dfs -help
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <loca
        [-count [-q] <path> ...]
        [-cp [-f] [-p] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
```

图 3-9 用户命令

### 3.4.3 HDFS 管理员命令

管理命令对于集群的管理员是很有用的，命令也比较多，例如平衡管理、缓存管理、DFS 管理、DataNode 管理等，对于用户记忆命令是个难题，同样用户也可以借助系统提供的帮助命令查看命令和命令参数，这给用户带来极大的方便，例如，查看 DFS 管理的命令帮助，只需要执行命令 hdfs dfsadmin -help，查看命令列表，如图 3-10 所示。

```
[root@simple01 bin]# hadoop dfsadmin -help
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

hadoop dfsadmin is the command to execute DFS administrative commands.
The full syntax is:

hadoop dfsadmin [-report] [-safemode <enter | leave | get | wait>]
        [-saveNamespace]
        [-rollEdits]
        [-restoreFailedStorage true|false|check]
        [-refreshNodes]
        [-setQuota <quota> <dirname>...<dirname>]
        [-clrQuota <dirname>...<dirname>]
        [-setSpaceQuota <quota> <dirname>...<dirname>]
        [-clrSpaceQuota <dirname>...<dirname>]
        [-finalizeUpgrade]
        [-rollingUpgrade [<query|prepare|finalize>]]
        [-refreshServiceAcl]
        [-refreshUserToGroupsMappings]
        [-refreshSuperUserGroupsConfiguration]
        [-refreshCallQueue]
        [-printTopology]
        [-refreshNamenodes datanodehost:port]
        [-deleteBlockPool datanodehost:port blockpoolId [force]]
        [-setBalancerBandwidth <bandwidth>]
        [-fetchImage <local directory>]
```

图 3-10　管理员命令

学习小贴士

如果想访问 http://192.168.0.201:50070/explorer.html#/上的文件 tmp，可能会碰到如下问题：

Permissiondenied:user=dr,who,access=READ_EXECUTE,inode="/tmp:root:supergroup:drwx------"

解决办法：在 hadoop 的安装目录$HADOOP_HOME 运行 bin/Hadoopdfs chmod −R 755 /tmp 即可解决。

## 【案例 3-2】HDFS Shell 基本命令(一)

（1）在任意目录下（需要配置 Hadoop 环境变量），执行命令 start-dfs.sh， 启动 HDFS 进程，如图 3-11 所示。

（2）启动 HDFS 之后，通过 jps 命令查看 HDFS 服务是否启动，执行命令 jps，查看运行的进程，如图 3-12 所示。

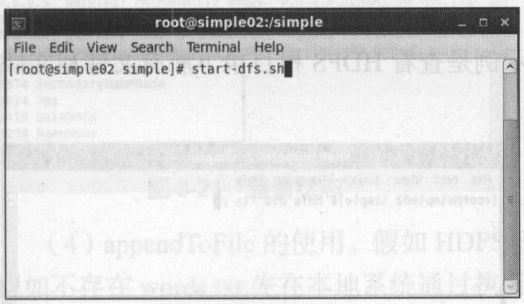



图 3-11　启动 HDFS

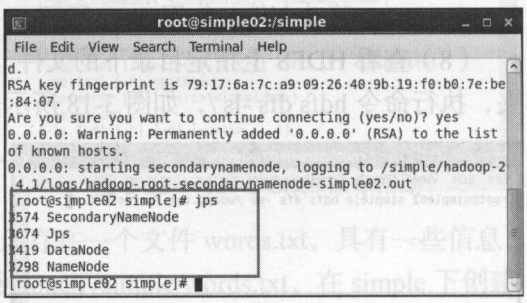



图 3-12　查看状态

（3）在任意目录下执行如下命令（配置 Hadoop 环境变量）：hdfs dfs -help，查看到所有 HDFS Shell 命令解释，如图 3-13 所示。

（4）在 simple 下执行命令 touch words.txt，新建 words.txt 文本，并对文本进行编译。上传本地文件“/simple/words.txt”到 HDFS。执行命令 hdfs dfs -put /simple/words.txt/，上传文件，如图 3-14 所示。

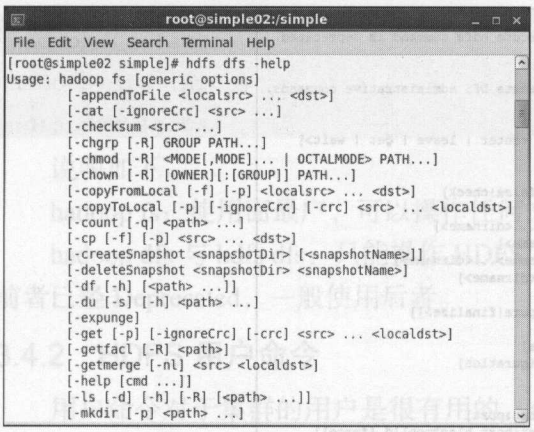



图 3-13　HDFS Shell 命令

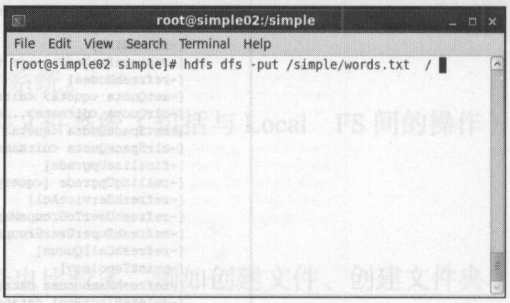



图 3-14　上传文件

（5）查看 HDFS 指定目录下的文件内容。执行命令 hdfs dfs -cat /words.txt，如图 3-15 所示。

（6）把 HDFS 根目录下的 words.txt 文件下载到/simple 目录下，执行命令 hdfs dfs -get /words.txt /simple/words.txt，下载文件，如图 3-16 所示。

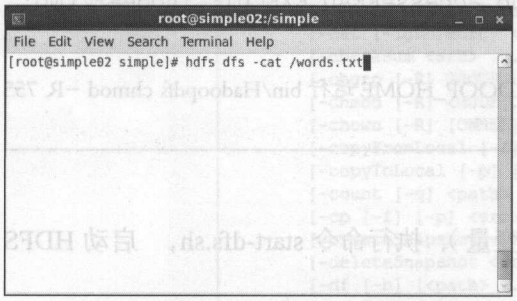



图 3-15　查看文件

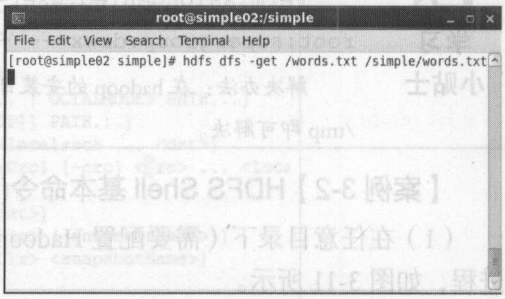



图 3-16　下载文件

（7）把 HDFS 上指定的文件移动到指定的 HDFS 位置，执行命令 hdfs dfs -mv /words.txt /wordscp.txt，移动文件，如图 3-17 所示。

（8）查看 HDFS 上指定目录下的文件，本示例是查看 HDFS 根目录下所有文件和文件夹，执行命令 hdfs dfs -ls /，如图 3-18 所示。

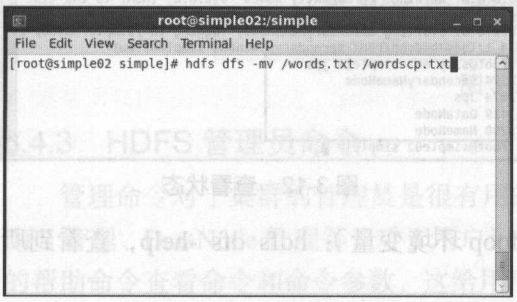



图 3-17　移动文件

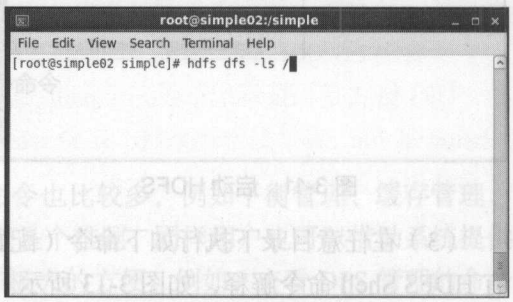



图 3-18　查看根目录文件

（9）在任意目录下执行命令 hadoop dfsadmin -help，查看 dfs 管理命令帮助，如图 3-19 所示。

【案例 3-3】HDFS Shell 基本命令(二)

（1）在任意目录下（需要配置 Hadoop 环境变量），执行命令 start-dfs.sh， 启动 HDFS 进程，如图 3-20 所示。

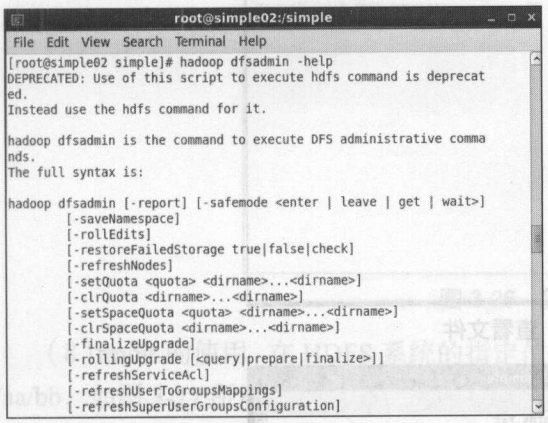



图 3-19　查看 dfs 管理命令

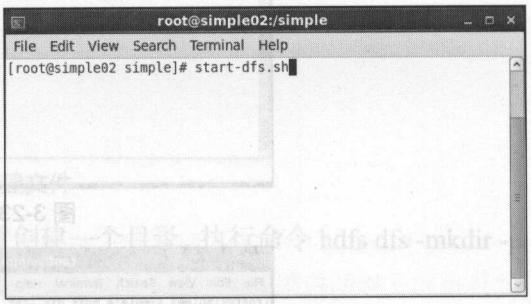



图 3-20　启动 HDFS 命令

（2）启动 HDFS 之后，通过 jps 查看 HDFS 进程是否启动，执行命令 jps ，查看进程状态，如图 3-21 所示。

（3）在任意目录下执行如下命令（配置 Hadoop 环境变量）：hdfs dfs -help，查看到所有 HDFS Shell 命令解释，如图 3-22 所示。

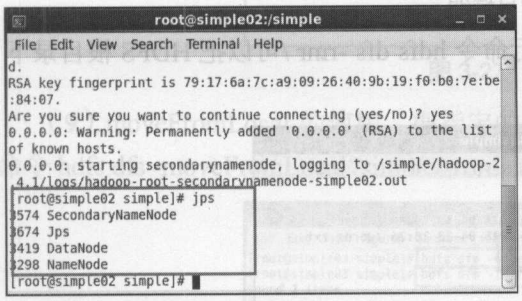



图 3-21　查看状态

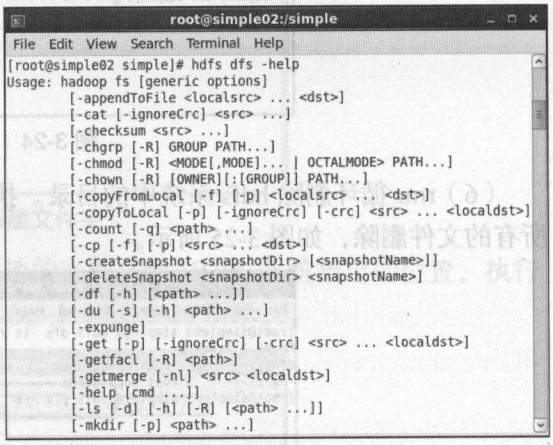



图 3-22　HDFS Shell 命令帮助

（4）appendToFile 的使用。假如 HDFS 上已经存在一个文件 words.txt，具有一些信息，假如不存在 words.txt,先在本地系统通过执行命令 touch /simple/words.txt，在 simple 下创建 words.txt 文件,并进行编译。然后通过 hdfs dfs -put /simple/words.txt /上传到 HDFS 根目录，如图 3-23 所示。

（5）在本地 simple 文件夹中创建 words.txt 文件,把指定的本地文件（/simple/words.txt）中的内容追加到 HDFS 系统的 words.txt 文件中。执行命令 hdfs dfs -appendToFile /simple/words.txt/words.txt。查看文件命令可以看出，HDFS 中的文件 words.txt 中的内容多一部分，如图 3-24 所示。

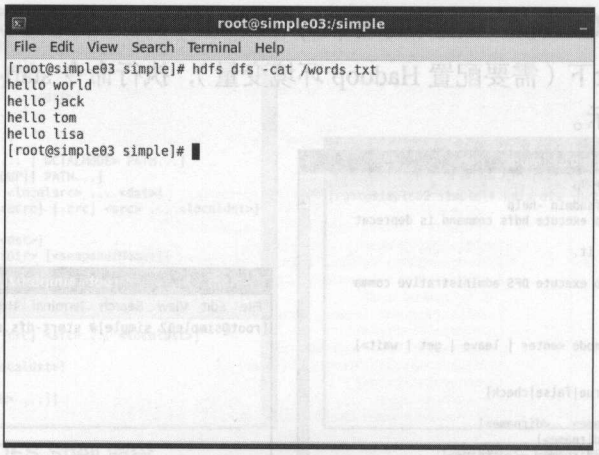



图 3-23　查看文件

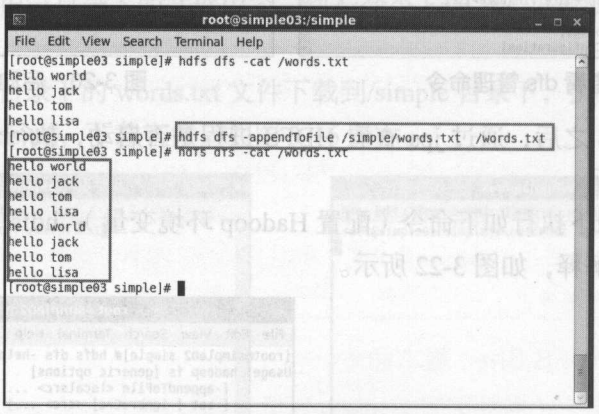



图 3-24　内容追加

（6）rmr 循环删除 hdfs 系统中的目录。执行命令 hdfs dfs -rmr /可以把 HDFS 根目录下所有的文件删除，如图 3-25 所示。

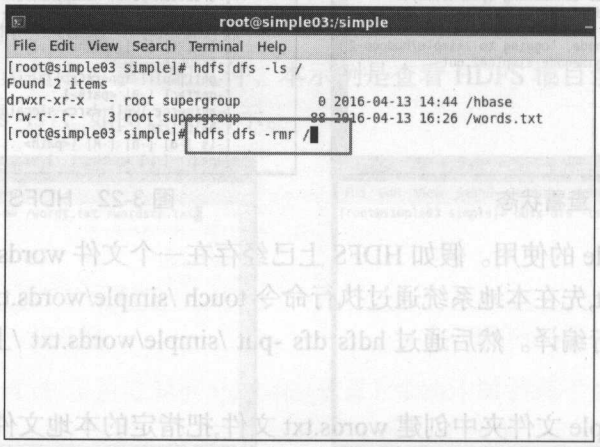



图 3-25　移除文件

（7）touchz 的使用。在 HDFS 系统指定的目录下创建一个文件。执行命令 hdfs dfs -touchz /newword.txt，创建一个文件，如图 3-26 所示。

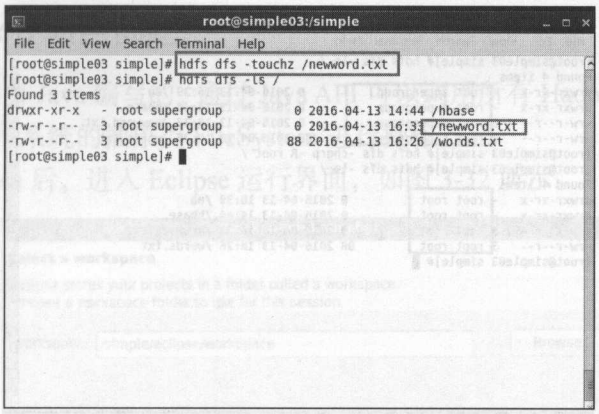



图 3-26 新建文件

（8）rmdir 的使用。在 HDFS 系统的指定位置创建一个目录，执行命令 hdfs dfs -mkdir -p /aa/bb，如图 3-27 所示。

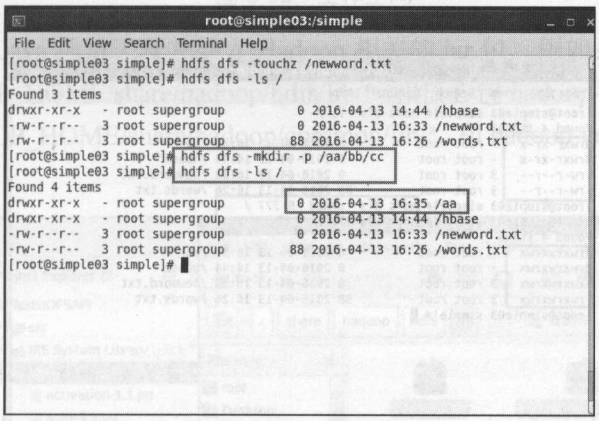



图 3-27 新建文件夹

（9）moveFromLocal 的使用。把指定的本地的文件移到 HDFS 系统指定的位置，执行命令 hdfs dfs -moveFromLocal /simple/words.txt /aa ，如图 3-28 所示。

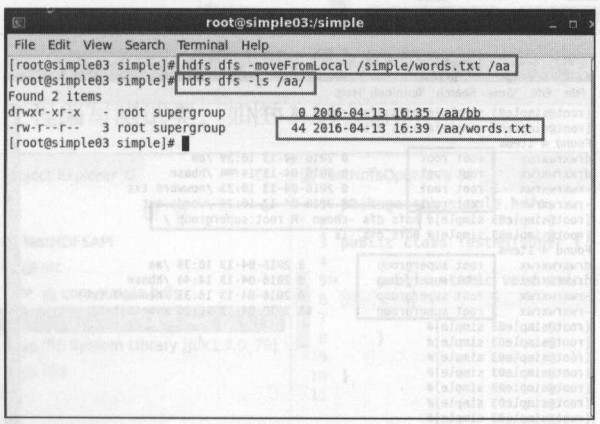



图 3-28 移到指定位置

（10）chgrp 的使用。修改 HDFS 系统中指定文件或文件夹的用户所属组，如图 3-29 所示。

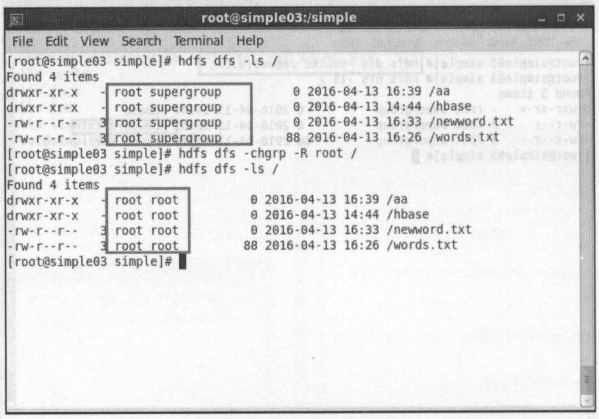



图 3-29　用户所属组

（11）chmod 的使用。改变指定目录文件的权限，如果指定 R 表示递归进行改变所有文件目录和文件的权限。用户必须是文件所有者或超级用户，执行命令 hdfs dfs -chmod -R 777 /，如图 3-30 所示。

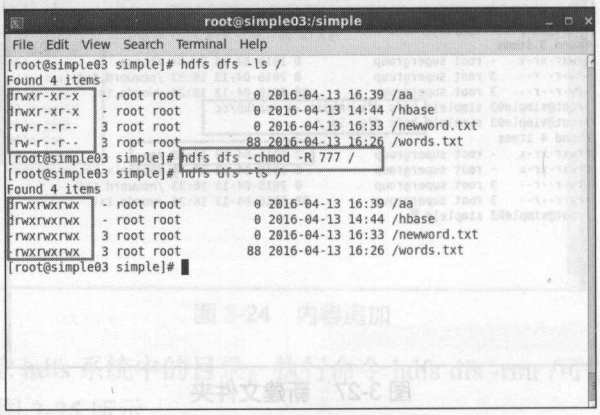



图 3-30　改变文件权限

（12）chown 的使用。改变文件的所有者，用户必须是超级用户，执行命令 hdfs dfs -chown -R root:supergroup /，如图 3-31 所示。

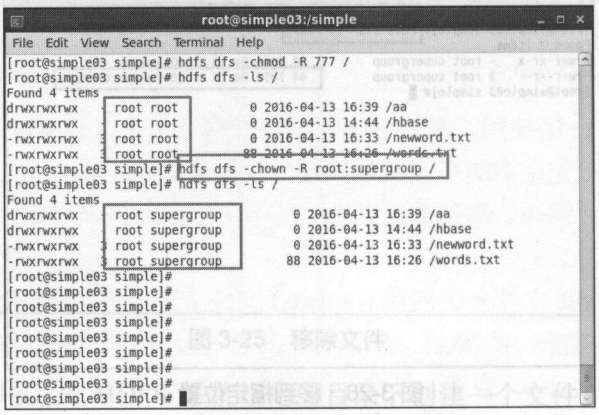



图 3-31　改变文件所有者

## 3.5 Hadoop 项目创建

Hadoop 本身是由 Java 编写的，通过 Java API 可以调用所有 Hadoop 文件系统的交互操作，通过 HDFS 文件系统的抽象类 FileSystem 进行操作。

首先启动 Eclipse 后，进入 Eclipse 运行界面，如图 3-32 所示。

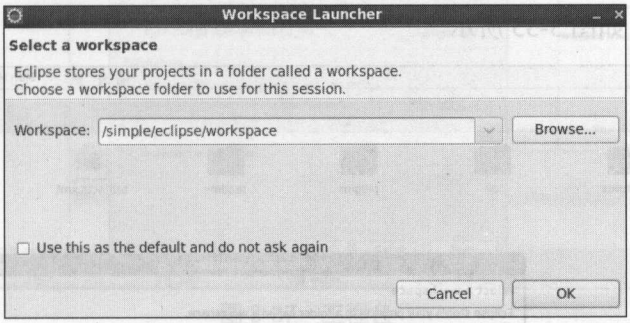



图 3-32 工作空间

然后创建一个 Java 项目工程并导入 Hadoop 相关的 jar 包。例如：

导入$HADOOP_HOME/share/hadoop/hdfs/lib/* 和核心包 hadoop-hdfs-2.4.1.jar。

导入$$HADOOP_HOME/share/hadoop/common/lib/* 和 hadoop-common-2.4.1.jar。

如图 3-33 所示。

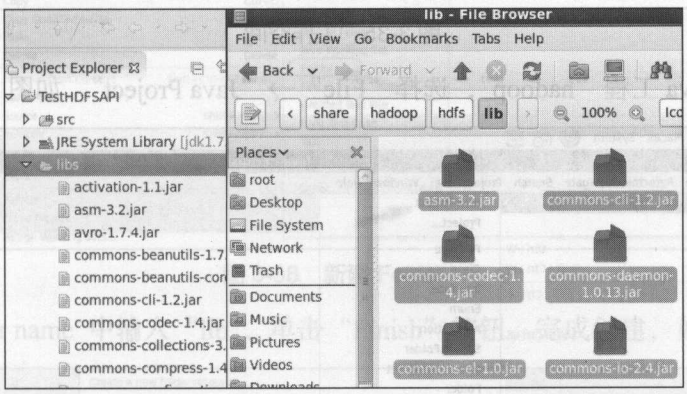



图 3-33 导入 jar 包

最后创建 Java 工程编写测试类，如图 3-34 所示。

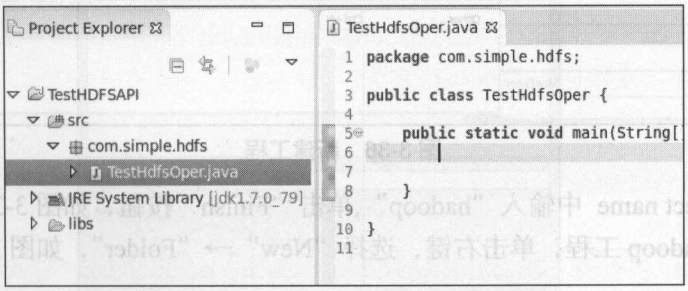



图 3-34 程序编译

【案例 3-4】Linux 下安装 Eclipse 及创建 HDFS 项目

（1）上传 Eclipse 压缩包“eclipse-java-mars-2-linux-gtk-x86_64.tar.gz” 到目录/soft 下。

（2）解压 Eclipse 到/usr。

tar -zxvf eclipse-java-mars-2-linux-gtk-x86_64.tar.gz -C /usr

（3）进入/usr/eclipse 中通过可视桌面打开 Eclipse，选择默认 Eclipse 工作表空间“/root/workspace”，如图 3-35 所示。

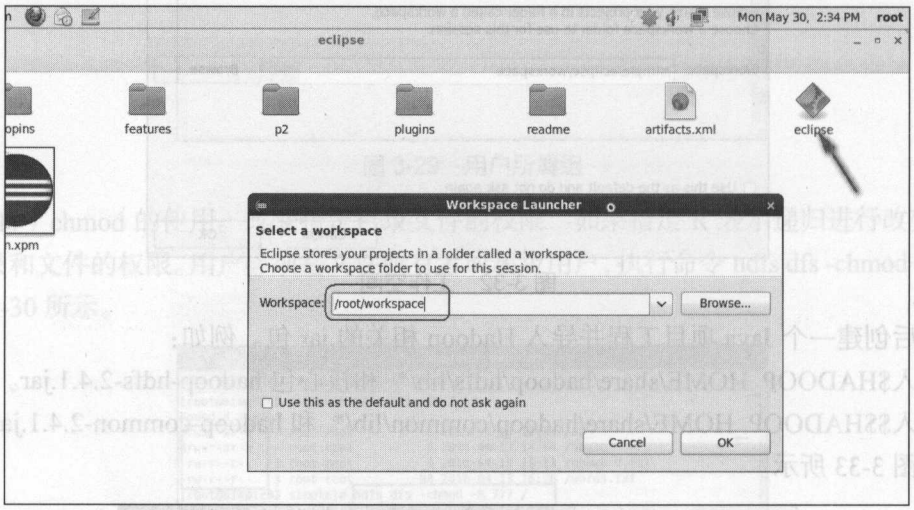



图 3-35 工作空间

（4）创建 Java 工程“hadoop”，选择“File”→“Java Project”，如图 3-36 所示。

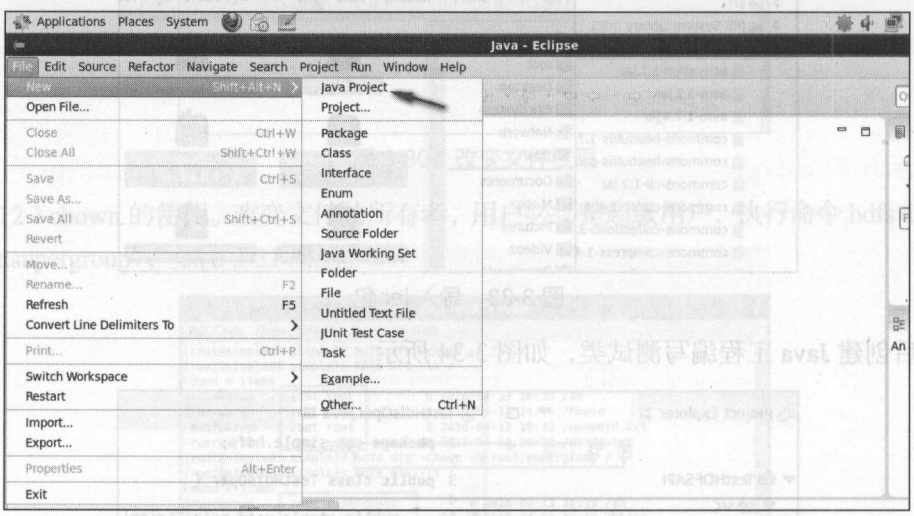



图 3-36 新建工程

（5）在 Project name 中输入“hadoop”,单击“Finish”按钮，如图 3-37 所示。

（6）选中 Hadoop 工程，单击右键，选择“New”→“Folder”，如图 3-38 所示。

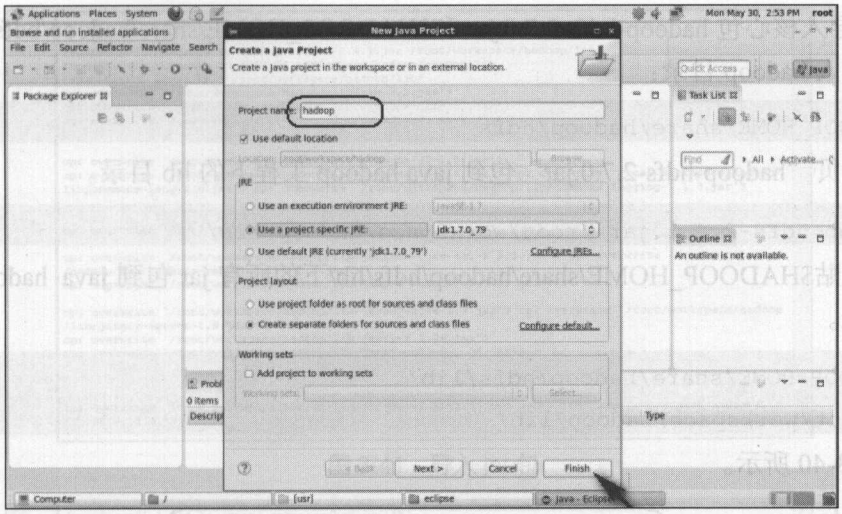



图 3-37　工程名

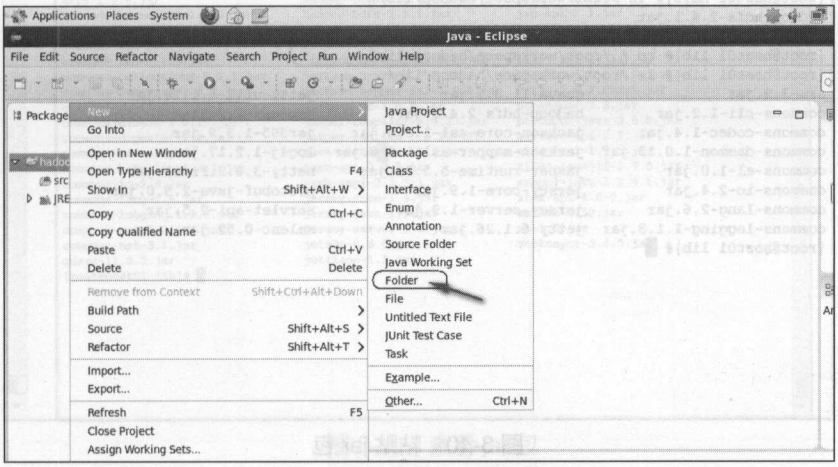



图 3-38　新建 Folder

（7）在 Folder name 中输入“lib”，单击“Finish”按钮，完成创建，如图 3-39 所示。

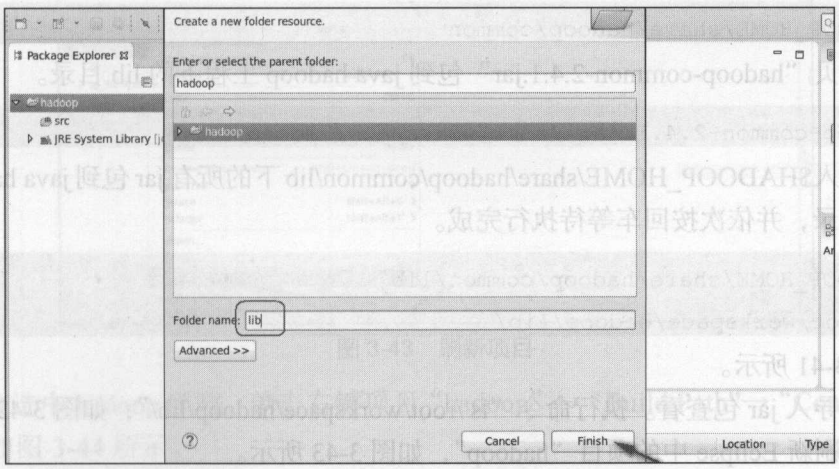



图 3-39　Folder 名

（8）导入核心包 hadoop-hdfs-2.7.0.jar 和$HADOOP_HOME/share/hadoop/hdfs/lib/*。

① 进入 hadoop 目录：

```
cd $HADOOP_HOME/share/hadoop/hdfs
```

② 拷贝“hadoop-hdfs-2.7.0.jar”包到 java hadoop 工程下的 lib 目录。

```
cp hadoop-hdfs-2.4.1.jar /root/workspace/hadoop/lib/
```

③ 粘贴$HADOOP_HOME/share/hadoop/hdfs/lib/下的所有 jar 包到 java hadoop 工程下的 lib 目录。

```
cd $HADOOP_HOME/share/hadoop/hdfs/lib/
cp * /root/workspace/hadoop/lib/
```

如图 3-40 所示。

```
[root@host01 lib]# cd $HADOOP_HOME/share/hadoop/hdfs
[root@host01 hdfs]# cp hadoop-hdfs-2.4.1.jar /root/workspace/hadoop/lib/
[root@host01 hdfs]# ls /root/workspace/hadoop/lib/
hadoop-hdfs-2.4.1.jar
[root@host01 hdfs]# cd $HADOOP_HOME/share/hadoop/hdfs/lib/
[root@host01 lib]# cp * /root/workspace/hadoop/lib/
[root@host01 lib]# ls /root/workspace/hadoop/lib/
asm-3.2.jar                 guava-11.0.2.jar            jetty-util-6.1.26.jar
commons-cli-1.2.jar         hadoop-hdfs-2.4.1.jar       jsp-api-2.1.jar
commons-codec-1.4.jar       jackson-core-asl-1.8.8.jar  jsr305-1.3.9.jar
commons-daemon-1.0.13.jar   jackson-mapper-asl-1.8.8.jar log4j-1.2.17.jar
commons-el-1.0.jar          jasper-runtime-5.5.23.jar   netty-3.6.2.Final.jar
commons-io-2.4.jar          jersey-core-1.9.jar         protobuf-java-2.5.0.jar
commons-lang-2.6.jar        jersey-server-1.9.jar       servlet-api-2.5.jar
commons-logging-1.1.3.jar   jetty-6.1.26.jar            xmlenc-0.52.jar
[root@host01 lib]#
```

图 3-40　粘贴 jar 包

（9）导入 hadoop-common-2.4.1.jar 和$HADOOP_HOME/share/hadoop/common/lib/*。

① 进入 hadoop 的 common 目录。

```
cd $HADOOP_HOME/share/hadoop/common
```

② 拷贝“hadoop-common-2.4.1.jar”包到 java hadoop 工程下的 lib 目录。

```
cp hadoop-common-2.4.1.jar /root/workspace/hadoop/lib/
```

③ 导入$HADOOP_HOME/share/hadoop/common/lib 下的所有 jar 包到 java hadoop 工程下的 lib 目录，并依次按回车等待执行完成。

```
cd $HADOOP_HOME/share/hadoop/common/lib
cp * /root/workspace/hadoop/lib/
```

如图 3-41 所示。

（10）导入 jar 包查看。执行命令“ls /root/workspace/hadoop/lib/”，如图 3-42 所示。

（11）刷新 Eclipse 中的项目“hadoop”，如图 3-43 所示。

```
[root@host01 lib]# clear
[root@host01 lib]# cd $HADOOP_HOME/share/hadoop/common
[root@host01 common]# cp hadoop-common-2.4.1.jar /root/workspace/hadoop/lib/
[root@host01 common]# cd $HADOOP_HOME/share/hadoop/common/lib
[root@host01 lib]# cp * /root/workspace/hadoop/lib/
cp: overwrite `/root/workspace/hadoop/lib/asm-3.2.jar'?
cp: overwrite `/root/workspace/hadoop/lib/commons-cli-1.2.jar'?
cp: overwrite `/root/workspace/hadoop/lib/commons-codec-1.4.jar'?


cp: overwrite `/root/workspace/hadoop/lib/commons-el-1.0.jar'?
cp: overwrite `/root/workspace/hadoop/lib/commons-io-2.4.jar'? cp: overwrite `/root/workspace/hadoop/
lib/commons-lang-2.6.jar'? cp: overwrite `/root/workspace/hadoop/lib/commons-logging-1.1.3.jar'?


cp: overwrite `/root/workspace/hadoop/lib/guava-11.0.2.jar'?

cp: overwrite `/root/workspace/hadoop/lib/jackson-core-asl-1.8.8.jar'? cp: overwrite `/root/workspace
/hadoop/lib/jackson-mapper-asl-1.8.8.jar'?
cp: overwrite `/root/workspace/hadoop/lib/jasper-runtime-5.5.23.jar'?

cp: overwrite `/root/workspace/hadoop/lib/jersey-core-1.9.jar'? cp: overwrite `/root/workspace/hadoop
/lib/jersey-server-1.9.jar'?
cp: overwrite `/root/workspace/hadoop/lib/jetty-6.1.26.jar'?
cp: overwrite `/root/workspace/hadoop/lib/jetty-util-6.1.26.jar'?



cp: overwrite `/root/workspace/hadoop/lib/jsp-api-2.1.jar'? cp: overwrite `/root/workspace/hadoop/lib
/jsr305-1.3.9.jar'?
```

图 3-41　导入 jar 包

```
[root@host01 lib]# ls /root/workspace/hadoop/lib/
activation-1.1.jar                hadoop-annotations-2.4.1.jar  jetty-6.1.26.jar
asm-3.2.jar                       hadoop-auth-2.4.1.jar         jetty-util-6.1.26.jar
avro-1.7.4.jar                    hadoop-common-2.4.1.jar       jsch-0.1.42.jar
commons-beanutils-1.7.0.jar       hadoop-hdfs-2.4.1.jar         jsp-api-2.1.jar
commons-beanutils-core-1.8.0.jar  httpclient-4.2.5.jar          jsr305-1.3.9.jar
commons-cli-1.2.jar               httpcore-4.2.5.jar            junit-4.8.2.jar
commons-codec-1.4.jar             jackson-core-asl-1.8.8.jar    log4j-1.2.17.jar
commons-collections-3.2.1.jar     jackson-jaxrs-1.8.8.jar       mockito-all-1.8.5.jar
commons-compress-1.4.1.jar        jackson-mapper-asl-1.8.8.jar  netty-3.6.2.Final.jar
commons-configuration-1.6.jar     jackson-xc-1.8.8.jar          paranamer-2.3.jar
commons-daemon-1.0.13.jar         jasper-compiler-5.5.23.jar    protobuf-java-2.5.0.jar
commons-digester-1.8.jar          jasper-runtime-5.5.23.jar     servlet-api-2.5.jar
commons-el-1.0.jar                java-xmlbuilder-0.4.jar       slf4j-api-1.7.5.jar
commons-httpclient-3.1.jar        jaxb-api-2.2.2.jar            slf4j-log4j12-1.7.5.jar
commons-io-2.4.jar                jaxb-impl-2.2.3-1.jar         snappy-java-1.0.4.1.jar
commons-lang-2.6.jar              jersey-core-1.9.jar           stax-api-1.0-2.jar
commons-logging-1.1.3.jar         jersey-json-1.9.jar           xmlenc-0.52.jar
commons-math3-3.1.1.jar           jersey-server-1.9.jar         xz-1.0.jar
commons-net-3.1.jar               jets3t-0.9.0.jar              zookeeper-3.4.5.jar
guava-11.0.2.jar                  jettison-1.1.jar
[root@host01 lib]#
```

图 3-42　查看 jar 包

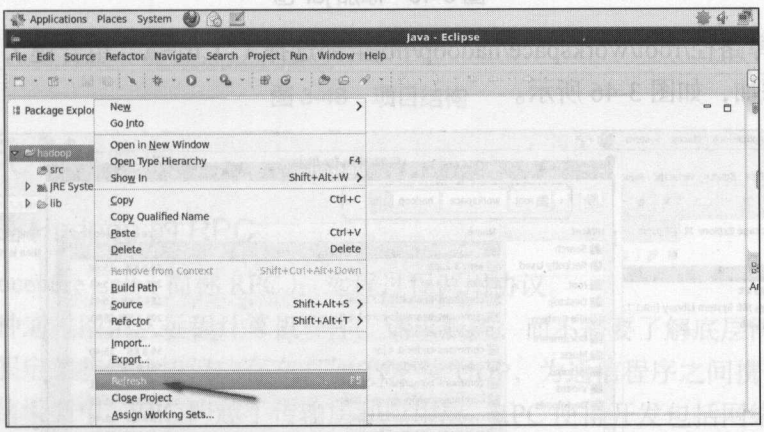



图 3-43　刷新项目

（12）选中 hadoop 工程，单击右键项目“hadoop”→“Build Path”→“Configure Build Path”，如图 3-44 所示。

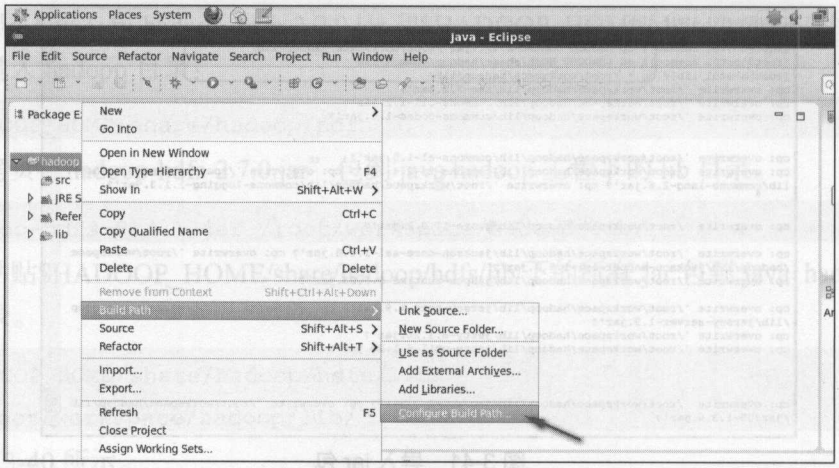



图 3-44　添加 jar 包

（13）选择“Add External JARs”，如图 3-45 所示。

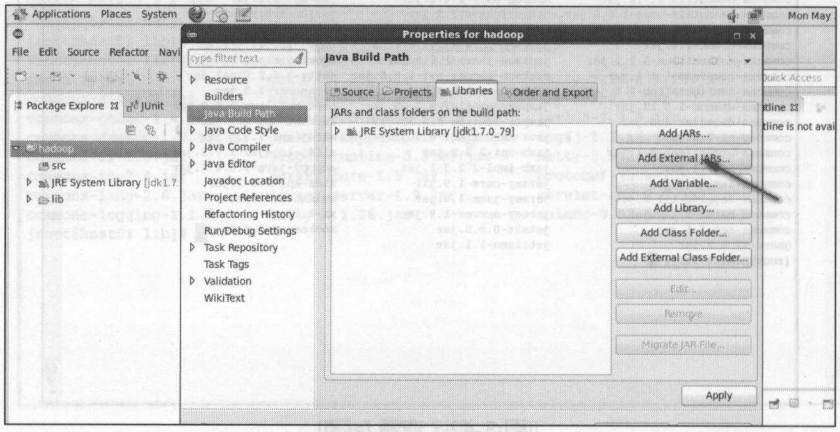



图 3-45　添加 jar 包

（14）选择路径/root/workspace/hadoop/lib,全部选择（“Ctrl+A”组合键），单击“OK”按钮，完成添加，如图 3-46 所示。

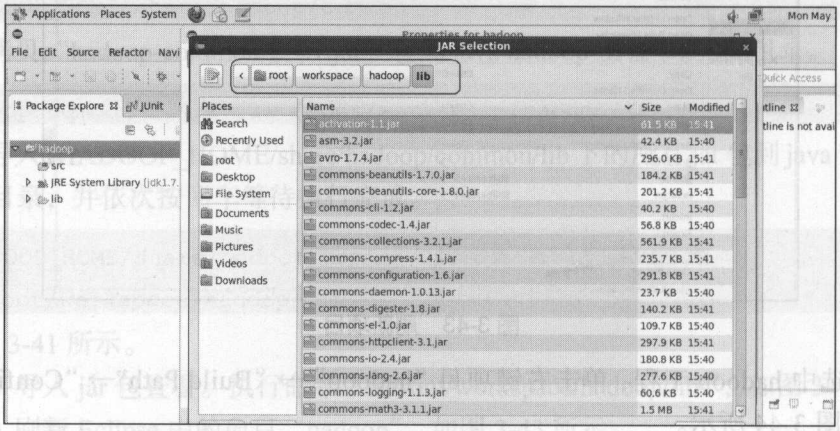



图 3-46　jar 包路径

（15）单击“OK”按钮，完成 jar 的添加，如图 3-47 所示。

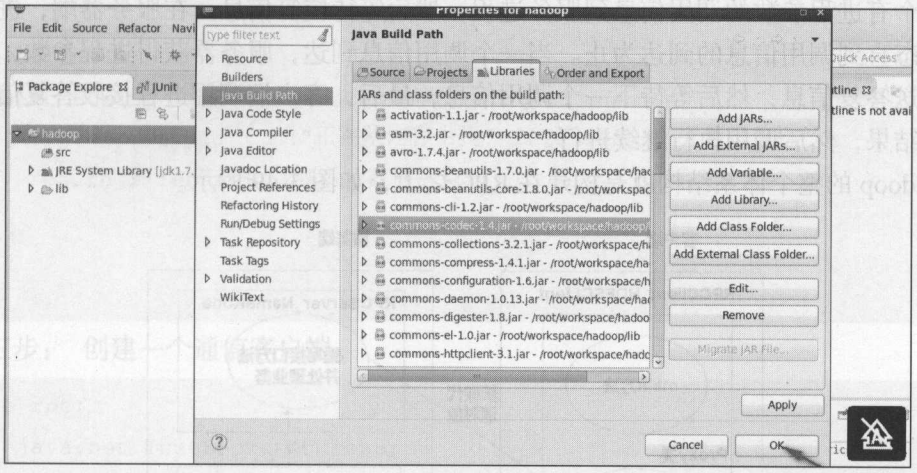



图 3-47　添加 jar

（16）项目结构，如图 3-48 所示。

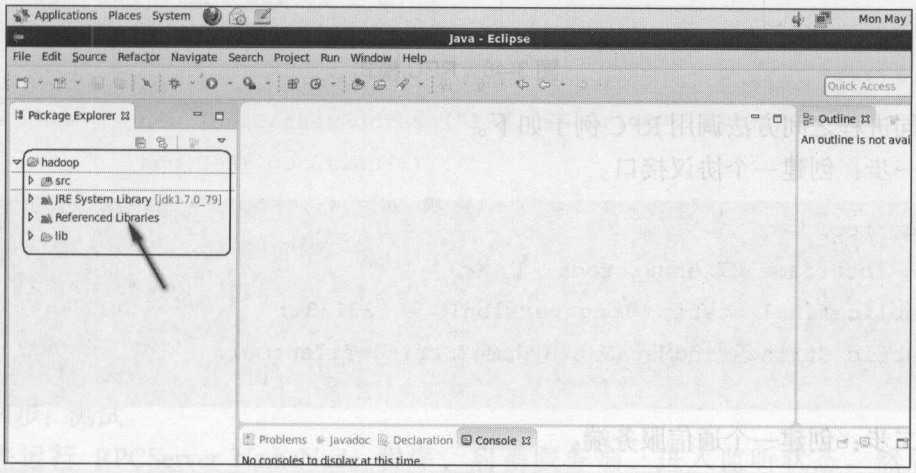



图 3-48　项目结构

## 3.6　RPC 通信原理

### 3.6.1　什么是 Hadoop 的 RPC

Remote Procedure Call（简称 RPC）：远程过程调用协议。

RPC 是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC 协议假定某些传输协议的存在，如 TCP 或 UDP，为通信程序之间携带信息数据。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

### 3.6.2　RPC 采用的模式

RPC 采用客户机/服务器模式。

请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

Hadoop 的整个体系结构就是构建在 RPC 之上，如图 3-49 所示。

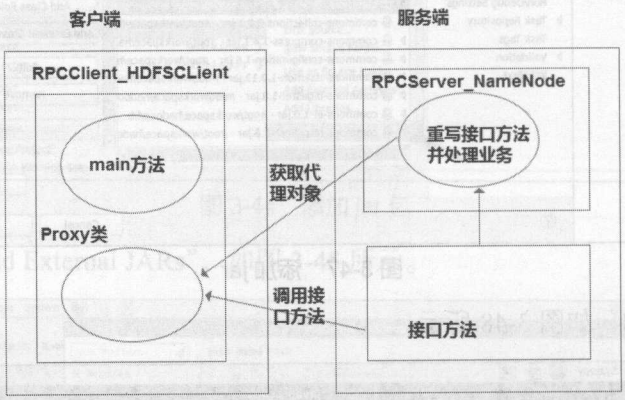



图 3-49　RPC 模式

不同进程之间方法调用 RPC 例子如下。

第一步：创建一个协议接口。

```
package rpc;
public interface ClientProtocal {
    public final static  long versionID = 1231231;
    public String findMetaDataByName(String filename);
}
```

第二步：创建一个通信服务端。

```
package rpc;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
import org.apache.hadoop.ipc.RPC.Server;
public class RPCServer_NameNode implements ClientProtocal{
    public static void main(String[] args) throws Exception{
        Server server = new RPC.Builder(new Configuration())
                .setInstance(new RPCServer_NameNode())

      .setProtocol(ClientProtocal.class).setBindAddress("192.168.0.110")
              .setPort(9123).build();
        server.start();
```

```
    }
    @Override
    public String findMetaDataByName(String filename){
        System.out.println("正在从内存中找"+filename+"的元数据信息");
        return filename+"找到后元数据信息";
    }
}
```

第三步： 创建一个通信客户端。

```
package rpc;
import java.net.InetSocketAddress;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
public class RPCClient_HDFSClient {
    public static void main(String[] args) throws Exception {
        ClientProtocal cp = RPC.getProxy(ClientProtocal.class, 123123,
                new InetSocketAddress("192.168.0.110", 9123),
                new Configuration());
        String msg = cp.findMetaDataByName("/words.txt");
        System.out.println(msg);
        RPC.stopProxy(cp);
    }
}
```

第四步：测试。

首先运行 RPCServer_NameNode 程序，启动服务端，进入监听状态。然后运行 RPCClient_HDFSClient 程序，启动客户端，连接服务端并运行结果，如图 3-50 所示。

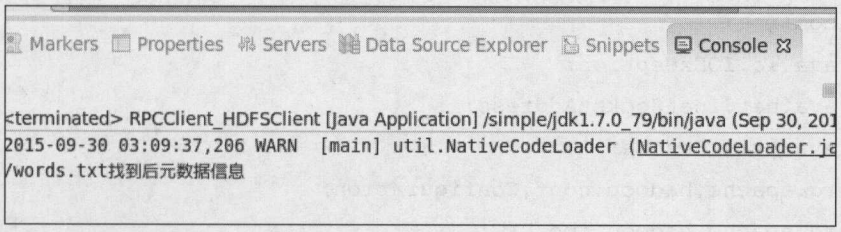



图 3-50 控制台

【案例 3-5】Hadoop 的 RPC 机制

（1）启动 Eclipse 并创建一个项目工程“hadoop”，在项目 src 上单击右键，选择“File”→“New”→“Package”，新建包名“com.rpc”，之后单击“Finish”按钮，如图 3-51 所示。

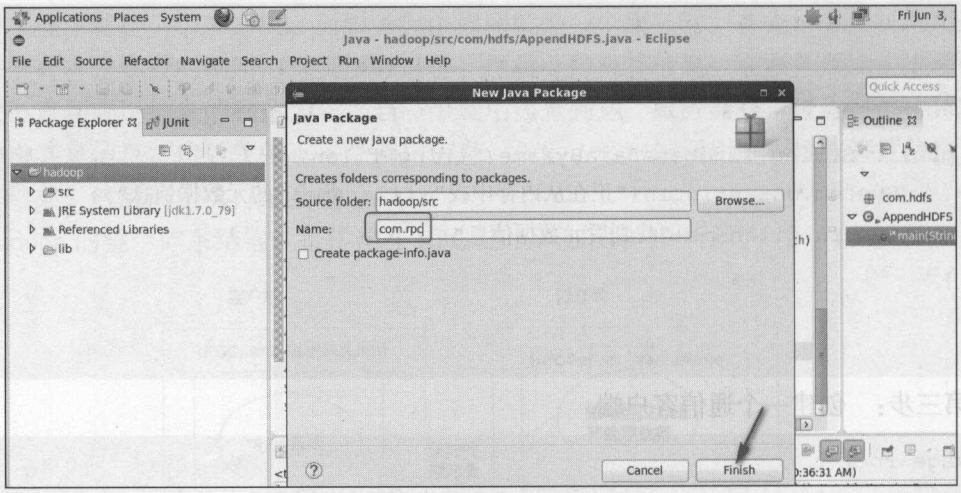



图 3-51　新建包名

（2）在包名“com.rpc”下创建接口 Isay,并编写接口代码如下。

```
package com.rpc;
/**
 * 定义 Isay 接口
*/
public interface Isay {
    /**
     * 定义 say 方法
     * @param userName 用户姓名
     * @return 回应语
     */
    public String say(String userName);
}
```

（3）在包名“com.rpc”下创建客户端类“SayRpcClient”并编写服务端程序。

```
package com.rpc;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.util.Scanner;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;

/**
 * 定义 RPC 通信 Client 端*
 */
public class SayRpcClient {
```

```
    private static Scanner sc;
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws Exception {
        //要连接服务端的通信地址及端口
        InetSocketAddress inetSocketAddress = new InetSocketAddress("127.0.0.1",
8000);
        //初始化 Rpc 代理
        Isay proxySay = RPC.getProxy(Isay.class, 13911112222L,inetSocketAddress,
new Configuration());
        System.out.println("请输入用户姓名（退出请输入 quit）：");
        //从控制台读取数据
        sc = new Scanner(System.in);
        while(sc.hasNext()){//判断输入
            String userName=sc.next();//读取输入数据
            if("quit".equals(userName)){//设定退出条件
                System.out.println("RPC client exited");
                return;
            }
            String s = proxySay.say(userName);//远程调用
            System.out.println(s);//输入 Server 返回的数据
            System.out.println("请再输入用户姓名（退出请输入 quit）：");
        }
    }
}
```

（4）在包名称“com.rpc”下创建服务端类“SayRpcServer”并编写服务端程序。

```
package com.rpc;
import java.io.IOException;
import org.apache.hadoop.HadoopIllegalArgumentException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
import org.apache.hadoop.ipc.RPC.Builder;
import org.apache.hadoop.ipc.RPC.Server;
/**
 * RPC server 端实现类
*/
```

```
public class SayRpcServer implements Isay {

    @Override
    public String say(String userName) {
        System.out.println("server received data->["+userName+"]");
        return "Hello! 【 "+userName+"】";
    }
    public static void  main (String args[]) throws HadoopIllegalArgument
Exception, IOException{
        //创建 Prc server builder 对象
        Builder builder = new RPC.Builder(new Configuration());
        //设置服务对象
        builder.setInstance(new SayRpcServer());
        //设定服务端地址
        builder.setBindAddress("127.0.0.1");
        //设置服务端端口
        builder.setPort(8000);
        //定义代理协议类型
        builder.setProtocol(Isay.class);
        //创建 Server
        Server sayRpcServer = builder.build();
        //启动 Server
        sayRpcServer.start();
        System.out.println("server started");
    }
}
```

（5）代码执行结果。

client 端，如图 3-52 所示。

```
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.lib.Mutabl
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
请输入用户姓名（退出请输入quit）：
Jack
Hello! 【Jack】
请再输入用户姓名（退出请输入quit）：
Tom
Hello! 【Tom】
请再输入用户姓名（退出请输入quit）：

水电费
Hello! 【水电费】
请再输入用户姓名（退出请输入quit）：

quit
RPC client exited
```

图 3-52　客户端

server 端，如图 3-53 所示。

```
log4j:WARN No appenders could be found for logger (org.apache.hadoop.ipc.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
server started
server received data->[Jack]
server received data->[Tom]
server received data->[水电费]
```

图 3-53 服务端

## 3.7 分布式文件系统操作类

FileSystem 是一个实现了文件系统的抽象类，继承自 org.apache.hadoop.conf.Configured，并实现了 Closeable 接口，可以适用于多种文件系统，如本地文件系统 file://、ftp、hdfs 等。如果要自己实现一个系统可以通过继承这个类，做相应的配置，并实现相应的抽象方法。

（1）public abstract FSDataInputStream open(Path f, int bufferSize)//读出一个文件的数据时调用。

```
public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://simple01:9000/");
        FileSystem fs = FileSystem.get(conf);
        Path path = new Path("/words.txt");
        FSDataInputStream fis = fs.open(path);
        byte b[] = new byte[200];
        int i = fis.read(b);
        System.out.println(new String(b,0,i));
}
```

（2）public abstract FSDataOutputStream create//写入文件的数据时调用。

```
public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://simple01:9000/");
        FileSystem fs = FileSystem.get(conf);
        FSDataOutputStream  fos = fs.create(new Path("/wordss.txt"));
        fos.writeChars("hello");
}
```

（3）public abstract FSDataOutputStream append//在文件末尾添加数据。

```
public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://simple01:9000/");
        FileSystem fs = FileSystem.get(conf);
        Path path = new Path("/words.txt");
        FSDataOutputStream fss = fs.append(path);
        fss.writeChars("i love you");
}
```

（4）public abstract boolean rename//文件重命名。

```
public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://simple01:9000/");
        FileSystem fs = FileSystem.get(conf);
        Path path = new Path("/words.txt");
        boolean flag = fs.rename(path, new Path("/mywords.txt"));
        System.out.println(flag);
}
```

（5）public abstract boolean delete//删除一个文件或目录。

（6）public abstract FileStatus[] listStatus//列出路径下对应的文件集。

```
public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://simple01:9000/");
        FileSystem fs = FileSystem.get(conf);
        Path path = new Path("/");
        RemoteIterator<LocatedFileStatus> list = fs.listFiles(path, true);
        while(list.hasNext()){
            System.out.println(list.next());
        }
}
```

（7）public abstract void setWorkingDirectory//设置工作目录。

public abstract Path getWorkingDirectory()

（8）public abstract boolean mkdirs//创建一个目录。

（9）public abstract FileStatus getFileStatus//获取文件或目录的元数据。

**【案例 3-6】Java API 操作——创建目录**

（1）查看服务状态。在命令终端，执行命令 jps，查看进程状态（查看 hadoop 服务是否已经启动，如未启动，启动服务），如图 3-54 所示。

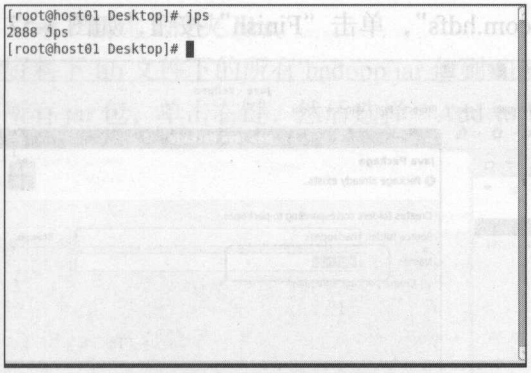



图 3-54 查看服务状态

（2）启动 Hadoop 进程。启动 Hadoop 服务可以通过一次性启动 Hadoop 所有进程，执行命令 start-all.sh，还可以通过单独启动，如图 3-55 所示。

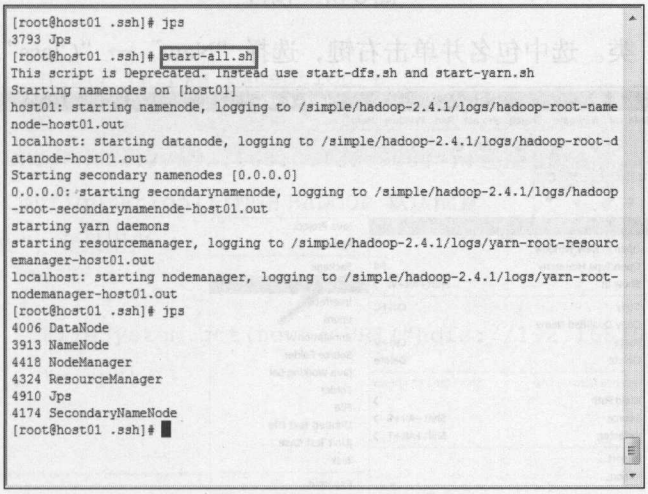



图 3-55 启动 Hadoop 服务

（3）打开 Eclipse 开发工具，单击 File 选择“New”→“Java Project”，新建名称为“Hadoop”的 Java 项目，单击右键“hadoop”项目，选择“New”→“Package”，如图 3-56 所示。

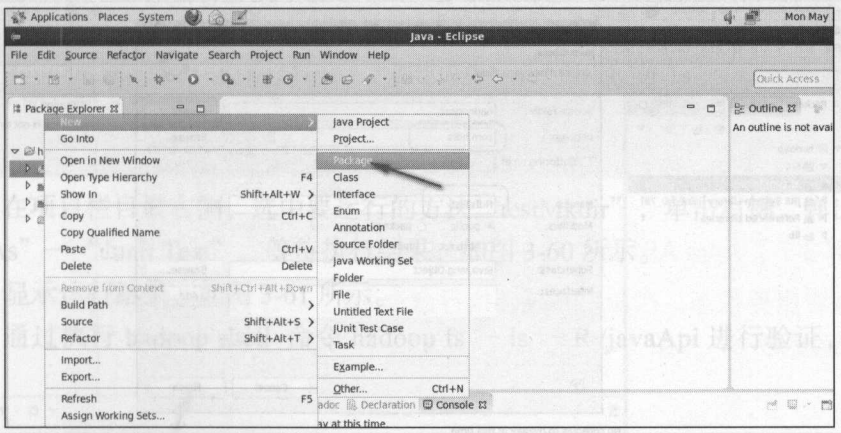



图 3-56 新建包名

（4）输入包名称“com.hdfs”，单击“Finish”按钮，如图 3-57 所示。

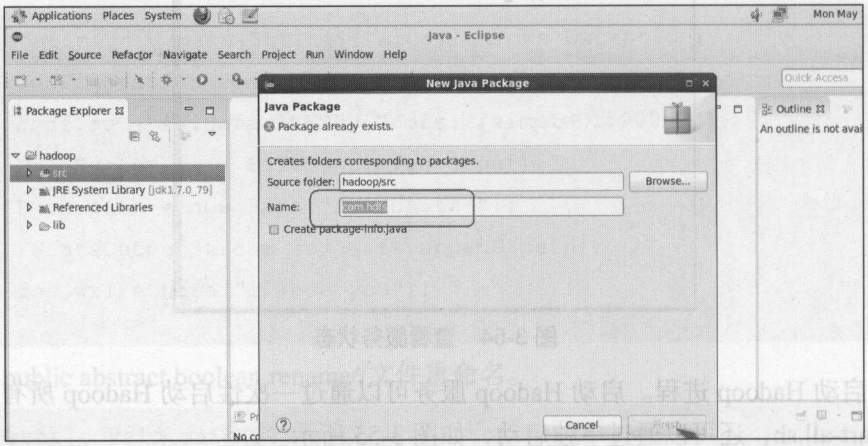



图 3-57　包名

（5）新建 java 类。选中包名并单击右键，选择“New”→“Class”，如图 3-58 所示。

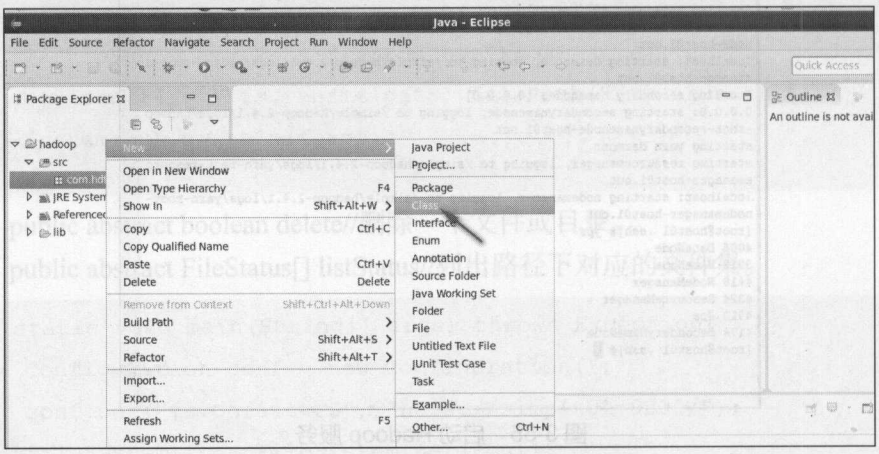



图 3-58　新建类

（6）在 name 项输入“ HdfsTest”类名称，单击“Finish”按钮完成，如图 3-59 所示。

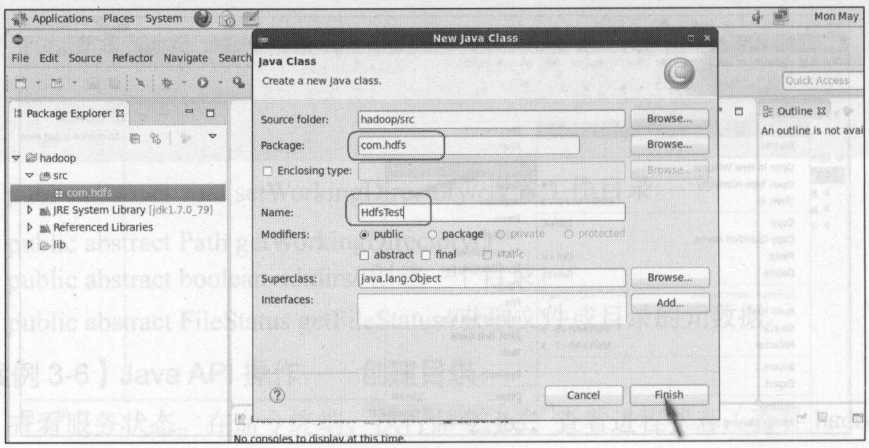



图 3-59　类名

（7）在创建的项目目录下创建文件夹 libs。

（8）拷贝桌面实验资料下 lib 文件下的所有 hadoop jar 包到 libs 文件夹下。

（9）选中 libs 下的所有 jar 包，单击右键，然后选择“Add to Build Path”即可把所有 jar 包添加到 path 环境中。

（10）编写程序。

```
package com.hdfs;
public class HdfsTest {
    // 获取 HADOOP FileSystem 对象
    private FileSystem fs = null;
    /**
     * 初始化环境变量
     */
    @Before
    public void init() throws Exception {
        /*
         * new URI("hdfs://192.168.0.131:9000"):连接 HDFS
         * new Configuration():使用 HADOOP 默认配置
         * "root": 登录用户
         */
        fs = FileSystem.get(new URI("hdfs://192.168.0.202:9000"),new
Configuration(), "root");
    }
    /**
     * 创建目录
     */
    @Test
    public void testMkdir() throws Exception {
        boolean flag = fs.mkdirs(new Path("/javaApi/mk/dir1/dir2"));
        System.out.println(flag ? "创建成功" : "创建失败");
    }
}
```

（11）在项目栏目最右侧，选中要运行的方法“testMkdir”，单击右键，弹出菜单，选择“Run As”→“Junit Test”，等待执行结果，如图 3-60 所示。

（12）显示运行结果，如图 3-61 所示。

（13）通过执行 hadoop shell 命令 hadoop fs -ls -R /javaApi 进行验证，如图 3-62 所示。

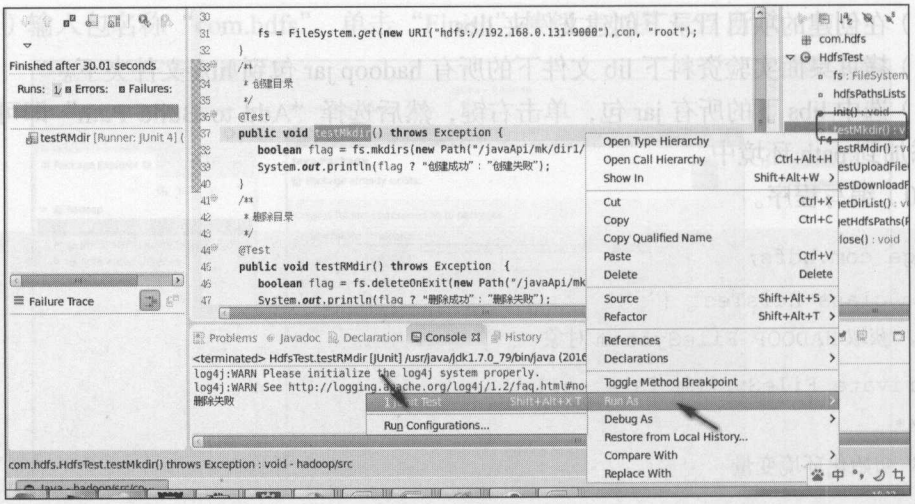



图 3-60　运行程序

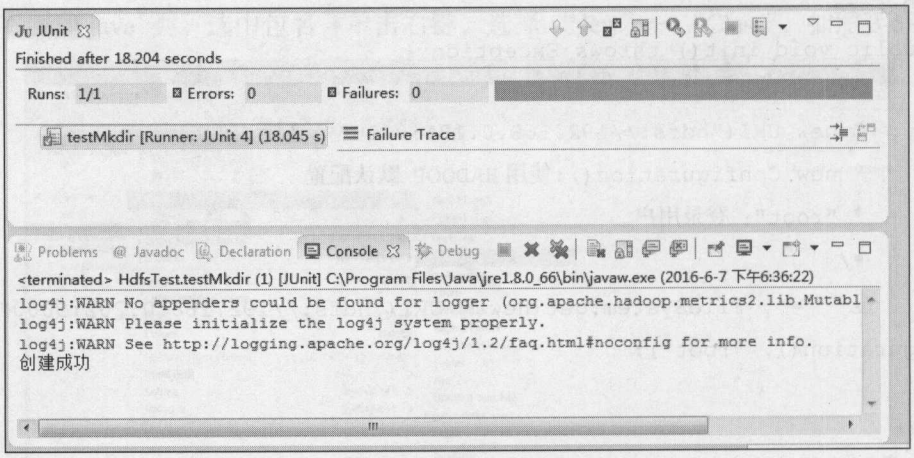



图 3-61　控制台信息

```
[root@host01 ~]# hadoop fs -ls /
Found 3 items
drwxr-xr-x   - root supergroup          0 2016-05-22 17:22 /hdfs
-rw-r--r--   1 root supergroup  153512879 2016-05-27 15:12 /jdk-7u79-linux-x64.tar.gz
drwxr-xr-x   - root supergroup          0 2016-05-22 17:21 /local
[root@host01 ~]# hadoop fs -ls /
Found 4 items
drwxr-xr-x   - root supergroup          0 2016-05-22 17:22 /hdfs
drwxr-xr-x   - root supergroup          0 2016-05-31 09:32 /javaApi
-rw-r--r--   1 root supergroup  153512879 2016-05-27 15:12 /jdk-7u79-linux-x64.tar.gz
drwxr-xr-x   - root supergroup          0 2016-05-22 17:21 /local
[root@host01 ~]# hadoop fs -ls -R /javaApi
drwxr-xr-x   - root supergroup          0 2016-05-31 09:32 /javaApi/mk
drwxr-xr-x   - root supergroup          0 2016-05-31 09:32 /javaApi/mk/dir1
drwxr-xr-x   - root supergroup          0 2016-05-31 09:32 /javaApi/mk/dir1/dir2
```

图 3-62　查看 HDFS

## 【案例 3-7】Java API 操作——上传文件

（1）查看服务状态。在命令终端，执行命令 jps（查看 Hadoop 服务是否已经启动，如未启动，启动服务），如图 3-63 所示。

```
[root@host01 Desktop]# jps
2888 Jps
[root@host01 Desktop]#
```

图 3-63　服务状态

（2）启动 Hadoop 服务。启动 Hadoop 服务可以通过一次性启动 Hadoop 所有服务，执行命令 start-all.sh，还可以通过单独启动，如图 3-64 所示。

```
[root@host01 .ssh]# jps
3793 Jps
[root@host01 .ssh]# start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [host01]
host01: starting namenode, logging to /simple/hadoop-2.4.1/logs/hadoop-root-name
node-host01.out
localhost: starting datanode, logging to /simple/hadoop-2.4.1/logs/hadoop-root-d
atanode-host01.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /simple/hadoop-2.4.1/logs/hadoop
-root-secondarynamenode-host01.out
starting yarn daemons
starting resourcemanager, logging to /simple/hadoop-2.4.1/logs/yarn-root-resourc
emanager-host01.out
localhost: starting nodemanager, logging to /simple/hadoop-2.4.1/logs/yarn-root-
nodemanager-host01.out
[root@host01 .ssh]# jps
4006 DataNode
3913 NameNode
4418 NodeManager
4324 ResourceManager
4910 Jps
4174 SecondaryNameNode
[root@host01 .ssh]#
```

图 3-64　启动 Hadoop 服务

（3）通过执行命令 touch /simple/HelloWorld.txt，在 simple 目录下创建 HelloWorld.txt 文件，如图 3-65 所示。

```
File  Edit  View  Search  Terminal  Help
[root@i-xqmic66m /]# touch /simple/HelloWorld.txt
[root@i-xqmic66m /]# cd /simple
[root@i-xqmic66m simple]# ls
eclipse                                hadoop-2.4.1  HelloWorld.txt  soft
eclipse-java-mars-1-linux-gtk-x86_64.tar.gz  Hadoop-2.4.1  jdk1.7.0_79
[root@i-xqmic66m simple]#
```

图 3-65　新建文件

（4）打开 Eclipse 开发工具，选择 File，选择“New”→“Java Project”，新建名称为“hadoop”的 Java 项目，单击右键“hadoop”项目，选择“New”→“Package”，如图 3-66 所示。

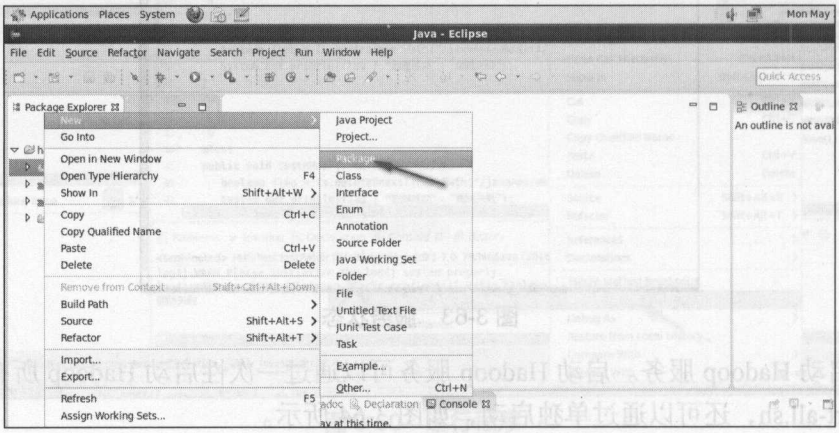



图 3-66　新建包名

（5）输入名称“com.hdfs”，单击“Finish”按钮，如图 3-67 所示。

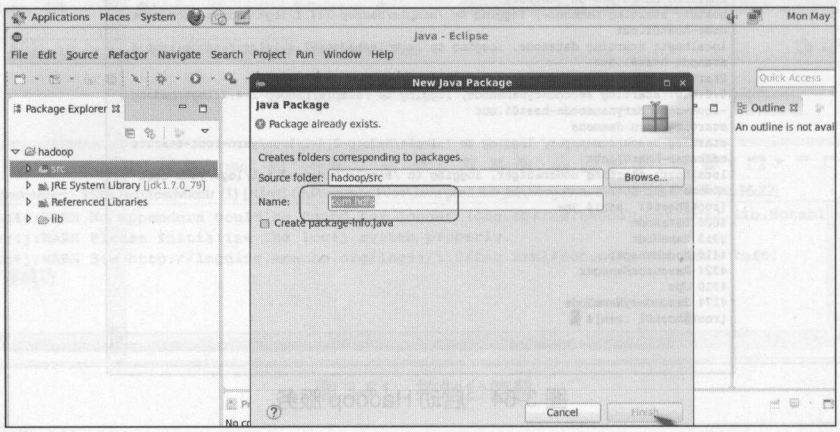



图 3-67　包名

（6）新建 java 类。选中包名并单击右键，选择“New”→“Class”，如图 3-68 所示。

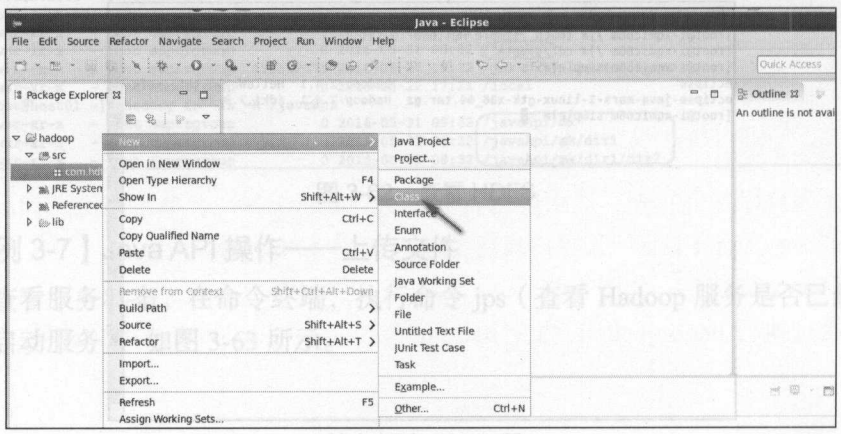



图 3-68　新建类

（7）在 name 项输入“ HdfsTest”类名称，单击“Finish”按钮完成，如图 3-69 所示。

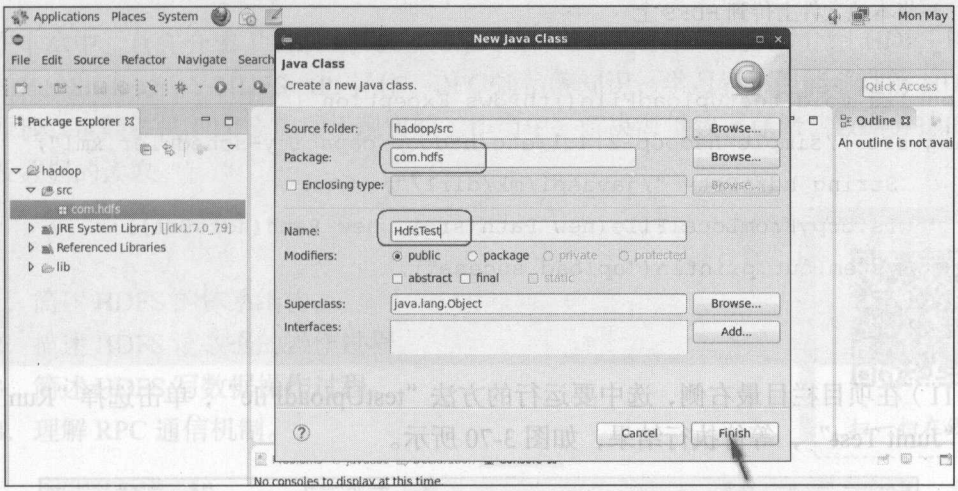



图 3-69 类名

（8）在创建的项目目录下创建文件夹 libs。

（9）拷贝桌面实验资料下 lib 文件下的所有 hadoop jar 包到 libs 文件夹下。

（10）选中 libs 下的所有 jar 包，单击右键，然后选择“Add to Build Path”即可把所有 jar 包添加到 path 环境中。

```
package com.hdfs;
public class HdfsTest {
    // 获取 HADOOP FileSystem 对象
    private FileSystem fs = null;
    private List<String> hdfsPathsLists;
    /**
     * 初始化环境变量
     */
    @Before
    public void init() throws Exception {
        /*
         * new URI("hdfs://192.168.0.201:9000"):HDFS
         * Configuration():使用 HADOOP 默认配置
         * "root": 登录用户
         */
        Configuration con = new Configuration();
        con.setBoolean("dfs.support.append", true);
        fs   =   FileSystem.get(new   URI("hdfs://192.168.0.201:9000"),con,
"root");
    }
```

```
    /**
     * 将本地文件上传到 HDFS 上
     */
    public void testUploadFile()throws Exception {
String src="/simple/hadoop-2.4.1/etc/hadoop/capacity-scheduler.xml";
        String hdfsDst= "/javaApi/mk/dir1/";
        fs.copyFromLocalFile(new Path(src), new Path(hdfsDst));
       System.out.println("upload sucess");
    }
}
```

（11）在项目栏目最右侧，选中要运行的方法“testUploadFile”，单击选择“Run As”，选择“Junit Test”，等待执行结果，如图 3-70 所示。

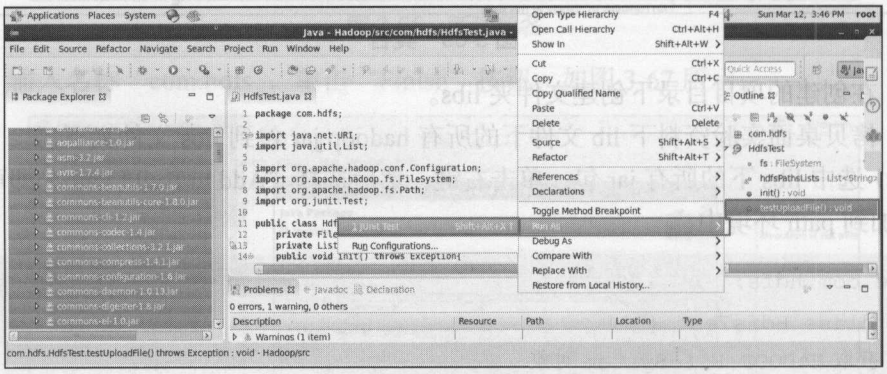



图 3-70　运行程序

（12）显示运行结果，如图 3-71 所示。

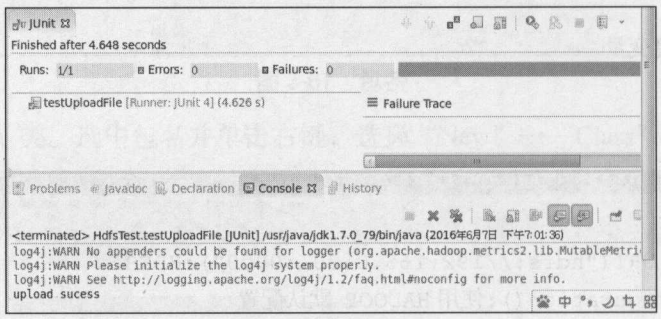



图 3-71　控制台信息

（13）通过 hadoop shell 命令进行验证，如图 3-72 所示。

```
[root@i-juzkaffy /]# hadoop fs -ls -R /javaApi
16/06/27 18:00:30 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
drwxr-xr-x   - root supergroup          0 2016-06-27 17:39 /javaApi/mk
-rw-r--r--   3 root supergroup          0 2016-06-27 17:39 /javaApi/mk/HelloWorl
d.txt
-rw-r--r--   3 root supergroup          0 2016-06-27 17:32 /javaApi/mk/dir1
[root@i-juzkaffy /]#
```

图 3-72　验证

## 本章小结

在本章中，首先介绍了 HDFS，让学习者了解 HDFS 是什么，然后介绍 HDFS 体系结构、HDFS Shell 命令、HDFS API 操作、RPC 通信等知识，学习者不但能够理解 HDFS 系统的原理，也能够把 HDFS 应用于现实的工作当中，解决海量数据存储、数据分拣的基本操作及数据的读取。

## 习题

1. 简述 HDFS 的体系结构。
2. 描述 HDFS 读数据的操作过程。
3. 简述 HDFS 写数据操作过程。
4. 理解 RPC 通信机制。

扫一扫在线测

# 第4章 计算系统 MapReduce

## 本章要点

- 了解 MapReduce 数据类型。
- 掌握 MapReduce 和 Yarn 的原理。
- 熟悉 Hadoop 数据类型接口。
- 理解 Hadoop 序列化。
- 熟悉 Hadoop 的执行过程。
- 掌握 MapReduce 接口类。

## 引　言

MapReduce 是 Hadoop 系统的核心组件，由 Google 的 MapReduce 系统经过演变而来，主要是解决海量大数据计算的，也是众多分布式计算模型中比较流行的一种，可以单独使用，一般配合 HDFS 一起使用。本章通过对 MapReduce 的概念、Hadoop 数据类型、Hadoop 序列化、Hadoop 的执行过程、MapReduce 常用接口类的学习，让学生深刻理解并学会运用 MapReduce 系统。

## 4.1 MapReduce 概念

### 4.1.1 MapReduce 简介

2004 年，Google 发表 MapReduce 的论文，MapReduce 主要应用于日志分析、海量数据的排序、索引计算等应用场景。后来道格·卡丁根据 Google 的论文 MapReduce 开发一个框架并将源代码贡献出来。

MapReduce 是一种思想，是一种分布式计算模型，由 Google 提出，主要用于搜索领域，解决离线海量数据的计算问题，但是不能实现对实时数据的分析和处理。对 Hadoop 来说，MapReduce 是一个分布式计算框架。归结起来就是“分而治之，迭代汇总”。就是把一个大的任务拆解开来，分成一系列小的任务并行执行，使得这些任务快速解决。MapReduce 解决问题的思路如图 4-1 所示。

MapReduce 由两个阶段组成。

map()：任务分解。

reduce()：结果汇总。

这两个函数的形参是 key、value 对，表示函数的输入信息。

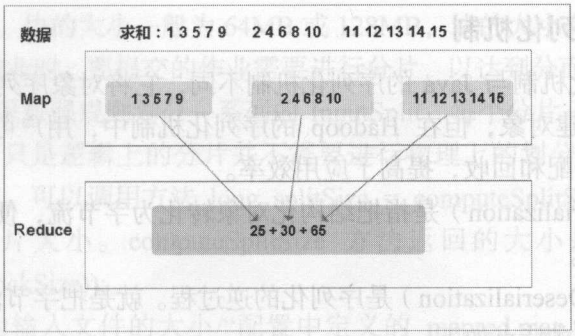



图 4-1 MapReduce 模型

### 4.1.2 MapReduce 数据类型与格式

Hadoop 框架是在 Java 的基础上进行开发的，Java 的数据类型在 Hadoop 中也能使用，但是具有一定的局限性，为了提高 Hadoop 对数据的处理能力，开发者在 Hadoop 研发中，增加了 Hadoop 特有的数据类型，这些数据类型通过实现接口 Writable 实现。如表 4-1 所示。

表 4-1 MapReduce 数据类型与格式

| Java 基本类型 | Writable 实现 | 序列化大小 |
|---|---|---|
| Boolean | BooleanWritable | 1 |
| byte | ByteWritable | 1 |
| int | IntWritable | 4 |
| | VintWritable | 1 ~ 5 |
| float | FloatWritable | 4 |
| long | LongWritable | 8 |
| | VlongWritable | 1 ~ 9 |
| double | DoubleWritable | 8 |

### 4.1.3 数据类型 Writable 接口

在 Hadoop 中 Java 基本类型都可以对应 Writable 类型的封装，Hadoop 将很多 Writable 类归入 org.apache.hadoop.io 包中，所有这些 Writable 类都是继承自 WritableComparable。也就是说，它们是可比较的。同时，它们都有 get()和 set()方法，用于获得和设置封装的值。

- Writable 接口，是根据 DataInput 和 DataOutput 实现的简单、有效的序列化对象。
- MapReduce 的任意 Key 和 Value 必须实现 Writable 接口。
- MapReduce 的任意 Key 必须实现 WritableComparable 接口。

学习小贴士

Text 一般认为它等价于 java.lang.String 的 Writable。针对 UTF-8 序列。

例如：

```
Text text=new Text("test");
IntWritable one = new IntWritable(1)
```

### 4.1.4 Hadoop 序列化机制

Hadoop 的序列化机制与 Java 的序列化机制不同，它将对象序列化到流中，Java 的序列化机制是不断地创建对象，但在 Hadoop 的序列化机制中，用户可以复用对象，这样就减少了 Java 对象的分配和回收，提高了应用效率。

（1）序列化（Serialization）是指把结构化对象转化为字节流，便于在网络上传输或写到磁盘进行永久存储。

（2）反序列化（Deserialization）是序列化的逆过程。就是把字节流转换为结构化对象。

Hadoop 序列化格式特点如下。

- 紧凑：高效使用存储空间。
- 快速：读写数据的额外开销小。
- 可扩展：可透明地读取老格式的数据。
- 互操作：支持多语言的交互。

Hadoop 序列化的作用如下。

- 序列化在分布式环境的两大作用：进程间通信，永久存储。
- Hadoop 节点间通信。

## 4.2 MapReduce 架构

Hadoop 中的 MapReduce 主要有两个版本：MRv1 和 MRv2。MRv1 包括 3 部分：MapReduce 编程模型、数据处理引擎和 MapReduce 运行环境（JobTrack 和 TaskTrack）。MapReduce 编程模型对任务抽象成 map 和 reduce 处理的任务。数据处理引擎负责任务运行的数据处理，包括数据分片、任务数据的输入输出。MapReduce 运行环境为程序的运行提供支持，如节点通信、任务分配和调度、资源管理等。MRv1 的弊端是 JobTrack 管理所有的任务，当任务过多时，JobTrack 的负载过重，造成系统不能正常运行。如图 4-2 所示。

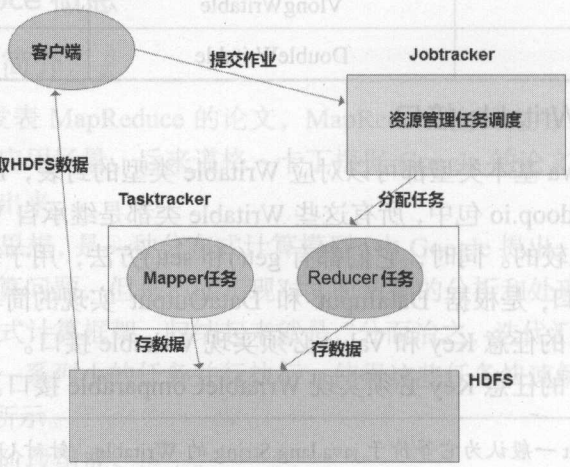



图 4-2 MapReduce 架构

### 4.2.1 数据分片

HDFS 存储系统中，为了存储引出块的概念，块是存储的最小单位，HDFS 上的文件都

是以块为单位存储的。块的大小一般为 64MB 或 128MB，块的大小有可能会变得更大。

Hadoop 在提交作业时，要提交的作业需要进行分片，以达到分而治之的目的，分片操作是根据源文件的情况按照规则划分一系列的 InputSplit，每个分片一般有一个 Mapper 进行处理，所谓的分片只是逻辑上的分片并不需要进行物理上的划分。具体的分片细节由 InputSplitFormat 指定，可以调用方法 long splitSize = computeSplitSize(goalSize, minSize, blockSize);以获取分片大小。computeSplitSize 方法返回的大小是 Math.max(minSize, Math.min(goalSize, blockSize));。

其中 goalSize 为输入文件的大小/“配置中定义的 mapred.map.tasks”值，minSize 为 mapred.min.split.size，blockSize 为 128MB，所以，这个算式为取分片大小不大于 block，并且不小于在 mapred.min.split.size 配置中定义的最小 Size。当某个分块分成均等的若干分片时，会有最后一个分片大小小于定义的分片大小，则该分片独立成为一个分片。

### 4.2.2 MapReduce 执行过程

为了更简单地理解 MapReduce 的执行过程，参考 MapReduce 的总体执行过程图，如图 4-3 所示。

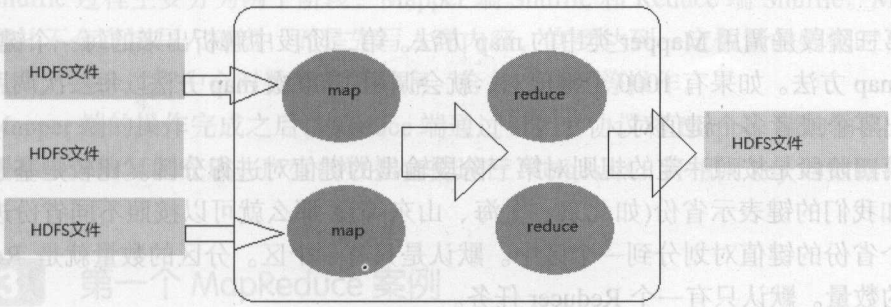



图 4-3 执行过程

MapReduce 运行的时候，通过 Mapper 运行的任务读取 HDFS 中的数据文件，然后调用自己的方法，处理数据，最后输出。Reducer 任务会接收 Mapper 任务输出的数据，作为自己的输入数据，调用自己的方法，最后输出到 HDFS 的文件中。

### 4.2.3 Mapper 执行过程

每个 Mapper 任务是一个 Java 进程，它会读取 HDFS 中的文件，解析成很多的键值对，经过我们覆盖的 map 方法处理后，转换为很多的键值对再输出。Mapper 接收<key,value>形式的数据并处理成<key,value>形式的数据，具体的处理过程用户可以定义。整个 Mapper 任务的处理过程又可以分为以下几个阶段，如图 4-4 所示。

（1）第一阶段是把输入文件按照一定的标准分片(InputSplit)，每个输入片的大小是固定的。默认情况下，输入片(InputSplit)的大小与数据块(Block)的大小是相同的。如果数据块(Block)的大小是默认值 64MB，输入文件有两个，一个是 32MB，一个是 72MB，那么小的文件是一个输入片，大文件会分为两个数据块 64MB 和 8MB，一共产生三个输入片。每一个输入片由一个 Mapper 进程处理。这里的三个输入片，会有三个 Mapper 进程处理。

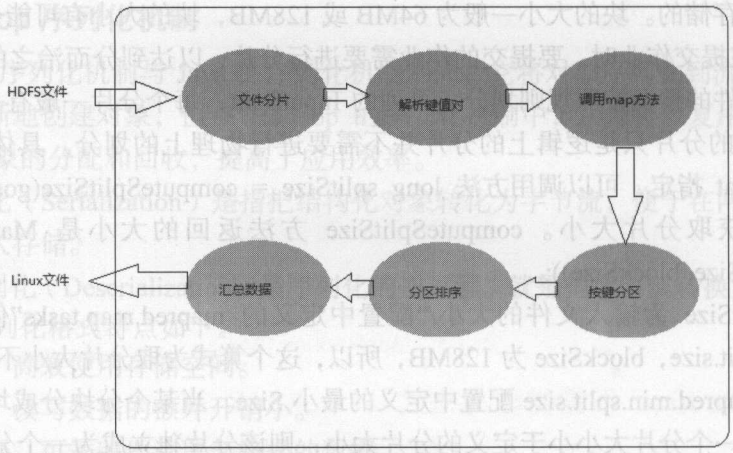



图 4-4　执行过程

（2）第二阶段是对输入片中的记录按照一定的规则解析成键值对。有个默认规则是把每一行文本内容解析成键值对。“键”是每一行的起始位置(单位是字节)，“值”是本行的文本内容。

（3）第三阶段是调用 Mapper 类中的 map 方法。第二阶段中解析出来的每一个键值对，调用一次 map 方法。如果有 1000 个键值对，就会调用 1000 次 map 方法。每一次调用 map 方法会输出零个或者多个键值对。

（4）第四阶段是按照一定的规则对第三阶段输出的键值对进行分区。比较是基于键进行的。比如我们的键表示省份(如北京、上海、山东等)，那么就可以按照不同省份进行分区，同一个省份的键值对划分到一个区中。默认是只有一个区。分区的数量就是 Reducer 任务运行的数量。默认只有一个 Reducer 任务。

（5）第五阶段是对每个分区中的键值对进行排序。首先，按照键进行排序，对于键相同的键值对，按照值进行排序。比如三个键值对<2,2>、<1,3>、<2,1>，键和值分别是整数。那么排序后的结果是<1,3>、<2,1>、<2,2>。如果有第六阶段，那么进入第六阶段；如果没有，直接输出到本地的 Linux 文件中。

（6）第六阶段是对数据进行归约处理，也就是 reduce 处理。键相等的键值对会调用一次 reduce 方法。经过这一阶段，数据量会减少。归约后的数据输出到本地的 Linux 文件中。本阶段默认是没有的，需要用户自己增加这一阶段的代码。

### 4.2.4　Reducer 执行过程

Reducer 任务接收 Mapper 任务的输出，归约处理后写入到 HDFS 中，可以分为如图 4-5 所示的几个阶段。

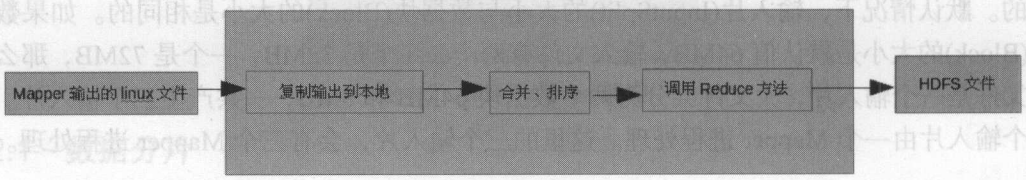



图 4-5　执行过程

（1）第一阶段是 Reducer 任务会主动从 Mapper 任务复制其输出的键值对。Mapper 任务可能会有很多，因此 Reducer 会复制多个 Mapper 的输出。

（2）第二阶段是把复制到 Reducer 的本地数据，全部进行合并，即把分散的数据合并成一个大的数据。再对合并后的数据排序。

（3）第三阶段是对排序后的键值对调用 reduce 方法,键相等的键值对调用一次 reduce 方法，每次调用会产生零个或者多个键值对。最后把这些输出的键值对写入到 HDFS 文件中。

注意

在整个 MapReduce 程序的开发过程中，我们最大的工作量是覆盖 map 函数和覆盖 reduce 函数。

### 4.2.5 Shuffle 过程

Shuffle 过程是从 Mapper 产生的输出数据开始，经过一系列的处理，最终成为 Reduce 的直接输入数据的整个过程，这一个过程也是 MapReduce 的核心过程。

Shuffle 过程主要分为两个阶段，Mapper 端 Shuffle 和 Reduce 端 Shuffle。Mapper 产生的数据并不会直接写入磁盘，而是先写入到内容，直到达到一定数量的数据时，才会写入到磁盘中。这个过程中会对数据进行排序、合并、分区等操作。

Mapper 端的操作完成之后，Reduce 端通过 HTTP 协议将 Mapper 端的输出 partition 赋值到缓存中，待复制完成，产生的数据会被进行 Mergesort，将相同 key 的数据排序集中到一起。

## 4.3 第一个 MapReduce 案例

（1）实现统计 HDFS 文件系统中的 words.txt 文件中的单词个数，可以通过执行命令 cat /simple/words.txt，查看文件的内容，如图 4-6 所示。

（2）经过 MapReduce 处理，对文本内容按行读取，过程如图 4-7 所示。

```
[root@hadoop11 ~]# cat /simple/words.txt
hello world
hello tom
hello jarry
hello kitter
you are good
think you
```

图 4-6 查看文件

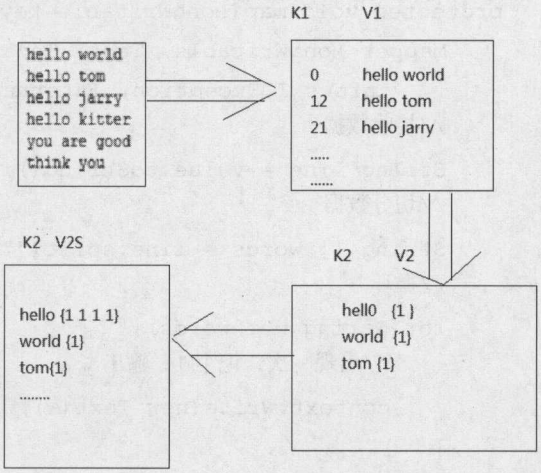



图 4-7 MapReduce 处理

（3）导入相关的 jar 包。

在 Hadoop 的项目中导入${HADOOP-HOME}/share/hadoop/mapreduce 红色选中部分的包，如图 4-8 所示。

```
[root@hadoop11 hadoop-2.4.1]# cd share/hadoop/mapreduce/
[root@hadoop11 mapreduce]# ls
hadoop-mapreduce-client-app-2.4.1.jar
hadoop-mapreduce-client-common-2.4.1.jar
hadoop-mapreduce-client-core-2.4.1.jar
hadoop-mapreduce-client-hs-2.4.1.jar
hadoop-mapreduce-client-hs-plugins-2.4.1.jar
hadoop-mapreduce-client-jobclient-2.4.1.jar
hadoop-mapreduce-client-jobclient-2.4.1-tests.jar
hadoop-mapreduce-client-shuffle-2.4.1.jar
hadoop-mapreduce-examples-2.4.1.jar
lib
lib-examples
sources
```

图 4-8　查看 jar

（4）在 Hadoop 的项目中导入${HADOOP-HOME}/share/hadoop/mapreduce/lib/*，如图 4-9 所示。

```
[root@hadoop11 mapreduce]# ls lib
aopalliance-1.0.jar              javax.inject-1.jar
asm-3.2.jar                      jersey-core-1.9.jar
avro-1.7.4.jar                   jersey-guice-1.9.jar
commons-compress-1.4.1.jar       jersey-server-1.9.jar
commons-io-2.4.jar               junit-4.10.jar
guice-3.0.jar                    log4j-1.2.17.jar
guice-servlet-3.0.jar            netty-3.6.2.Final.jar
hadoop-annotations-2.4.1.jar     paranamer-2.3.jar
hamcrest-core-1.1.jar            protobuf-java-2.5.0.jar
jackson-core-asl-1.8.8.jar       snappy-java-1.0.4.1.jar
jackson-mapper-asl-1.8.8.jar     xz-1.0.jar
[root@hadoop11 mapreduce]#
```

图 4-9　导入 jar

（5）创建 WcMapper，如下所示。

```
public class WcMapper extends Mapper<LongWritable, Text, Text, LongWritable>{
    @Override
    protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, Text, LongWritable>.Context context)
            throws IOException, InterruptedException {
        //接收数据
        String line = value.toString();
        //切分数据
        String [] words = line.split("");
        //循环
        for(String w : words){
            //出现一次，记个 1，输出
            context.write(new Text(w), new LongWritable(1));
        }
    }
}
```

（6）创建 WcReducer，如下所示。

```
public class WcReducer extends Reducer<Text, LongWritable, Text,
LongWritable>{
    @Override
    protected void reduce(Text key, Iterable<LongWritable> v2s,
            Reducer<Text, LongWritable, Text, LongWritable>.Context context)
            throws IOException, InterruptedException {
            //定义一个计算器
            long counter = 0;
            //循环 v2s
            for (LongWritable v : v2s) {
                counter +=v.get();
            }
            //输出
            context.write(key, new LongWritable(counter));
    }
}
```

（7）创建 WordCount，如下所示。

```
public class WordCount {
    public static void main(String[] args) throws IOException, ClassNotFound
Exception, InterruptedException {
        Job job = Job.getInstance(new Configuration());
        //注意：加载 main 方法所在的类
        job.setJarByClass(WordCount.class);
        //设置 Mapper 与 Reducer 的类
        job.setMapperClass(WcMapper.class);
        job.setReducerClass(WcReducer.class);
        FileInputFormat.setInputPaths(job,
new Path("hdfs://192.168.0.2:9000/words.txt"));
        FileOutputFormat.setOutputPath(job,
new Path("hdfs://192.168.0.2:9000/output"));
        //设置输入和输出的相关属性
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        job.waitForCompletion(true); //重要必写
    }
}
```

（8）打包成 jar file 上传到服务器运行测试。

在项目中单击右键“export”，如图 4-10 所示。

（9）单击“Next”按钮,选择 jar file 导出的位置，如图 4-11 所示。

（10）单击“Next”按钮，然后再继续单击“Next”按钮，选择 main 函数的测试类，并单击“OK”按钮,完毕，如图 4-12 所示。

（11）把 wc.jar 上传到服务器的指定目录进行测试，如图 4-13 所示。

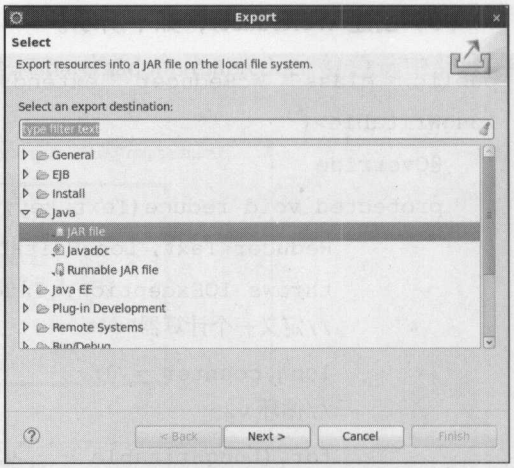



图 4-10　打包项目

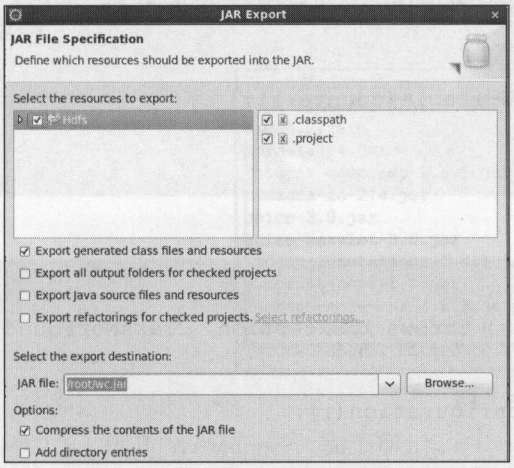



图 4-11　jar 位置

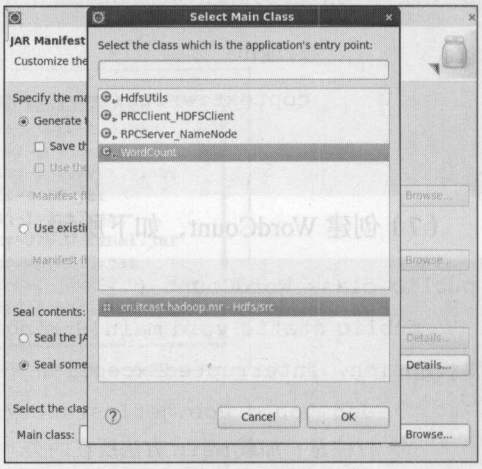



图 4-12　选择类

```
[root@hadoop11 sbin]# ls /root
anaconda-ks.cfg  Documents     install.log.syslog  Pictures   Videos
derby.log        Downloads     metastore_db        Public     WC.jar
Desktop          install.log   Music               Templates
[root@hadoop11 sbin]# cd /root
```

图 4-13　上传 jar

在命令行中输入 hadoop jar wc.jar 即可运行，也可以直接在本地模式下运行，需要把 Yarn 下的所有 jar 包导入工程，需要在输入、输出里加入前缀 file:///，如下所示。

```
Job job = Job.getInstance(new Configuration());
job.setJarByClass(WordCount.class);
job.setMapperClass(WCMapper.class);
job.setReducerClass(WCReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
FileInputFormat.setInputPaths(job, new Path("file:///simple/words.txt"));
FileOutputFormat.setOutputPath(job, new Path("file:///simple/wc3"));
job.waitForCompletion(true);
```

（12）运算后结果如图 4-14 所示。

```
[root@hadoop11 ~]# hdfs dfs -cat /output14/part-r-00000
are     1
good    1
hello   4
jarry   1
kitter  1
think   1
tom     1
world   1
you     2
```

图 4-14 查看内容

## 4.4 MapReduce 接口类

### 4.4.1 MapReduce 输入的处理类

#### 1. FileInputFormat

FileInputFormat 是所有以文件作为数据源的 InputFormat 实现的基类，FileInputFormat 保存作为 job 输入的所有文件，并实现了对输入文件计算 splits 的方法。至于获得记录的方法是由不同的子类 TextInputFormat 进行实现的，InputFormat 的常用方法有 createRecordReader 和 getSplits，如图 4-15 所示。

| Method Summary | |
|---|---|
| abstract RecordReader<K, V> | createRecordReader(InputSplit split, TaskAttemptContext context)<br>Create a record reader for a given split. |
| abstract List<InputSplit> | getSplits(JobContext context)<br>Logically split the set of input files for the job. |

图 4-15 Method Summary

InputFormat 负责处理 MapReduce 的输入部分，主要有三个作用:验证作业的输入是否规范,把输入文件切分成 InputSplit,提供 RecordReader 的实现类,把 InputSplit 读到 Mapper 中进行处理。InputFormat 类的层次结构，如图 4-16 所示。

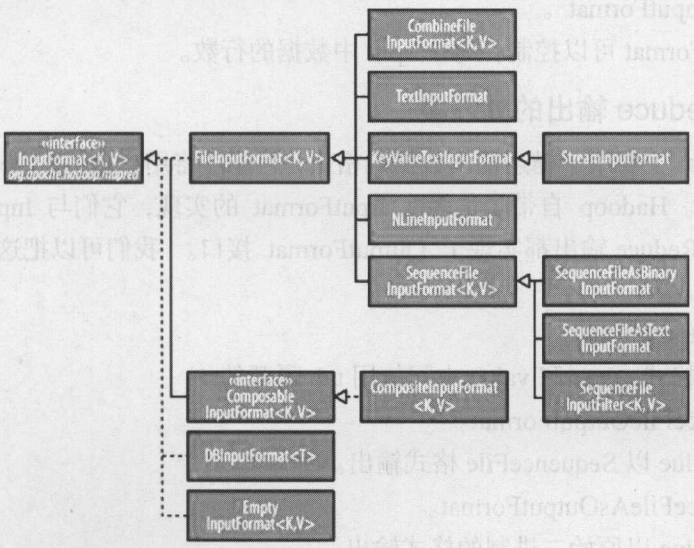



图 4-16 层次结构

2. InputSplit

在执行 MapReduce 之前，原始数据被分割成若干 split，每个 split 作为一个 map 任务的输入，在 map 执行过程中 split 会被分解成一个个记录（key-value 对），map 会依次处理每一个记录。

FileInputFormat 只划分比 HDFS Block 大的文件，所以 FileInputFormat 划分的结果是这个文件或者是这个文件中的一部分。

如果一个文件的大小比 Block 小，将不会被划分，这也是 Hadoop 处理大文件的效率要比处理很多小文件的效率高的原因。

当 Hadoop 处理很多小文件（文件大小小于 hdfs block 大小）的时候，由于 FileInputFormat 不会对小文件进行划分，所以每一个小文件都会被当作一个 split 并分配一个 map 任务，导致效率低下。

例如，一个 1GB 的文件，会被划分成 16 个 64MB 的 split，并分配 16 个 map 任务处理，而 10000 个 100KB 的文件会被 10000 个 map 任务处理。

除了 FileInputFormat 类,控制数据的输入格式,还有其他常见的 API 类:

（1）TextInputFormat。

TextInputFormat 是默认的处理类，处理普通文本文件。文件中每一行作为一个记录，它将每一行在文件中的起始偏移量作为 key，每一行的内容作为 value。默认以\n 或回车作为一行记录。

（2）CombineFileInputFormat。

相对于大量的小文件来说，hadoop 更合适处理少量的大文件。

CombineFileInputFormat 可以缓解这个问题，它是针对小文件而设计的。

（3）KeyValueTextInputFormat。

当输入数据的每一行是两列，并用 tab 分离的形式的时候，KeyValueTextInputFormat 处理这种格式的文件非常适合。

（4）NLineInputFormat 。

NLineInputFormat 可以控制在每个 split 中数据的行数。

### 4.4.2 MapReduce 输出的处理类

OutputFormat 主要用于描述输出数据的格式，它能够将用户提供的 key/value 对写入特定格式的文件中。Hadoop 自带了很多 OutputFormat 的实现，它们与 InputFormat 实现相对应，所有 MapReduce 输出都实现了 OutputFormat 接口。 我们可以把这些实现接口类分为以下几种类型。

（1）TextOutputFormat。

默认的输出格式，key 和 value 中间值用 tab 隔开的。

（2）SequenceFileOutputFormat。

将 key 和 value 以 SequenceFile 格式输出。

（3）SequenceFileAsOutputFormat。

将 key 和 value 以原始二进制的格式输出。

（4）MapFileOutputFormat。

将 key 和 value 写入 MapFile 中。由于 MapFile 中的 key 是有序的，所以写入的时候必须保证记录是按 key 值顺序写入的。

（5）MultipleOutputFormat。

默认情况下一个 reducer 会产生一个输出，但是有些时候我们想一个 reducer 产生多个输出，MultipleOutputFormat 和 MultipleOutputs 可以实现这个功能。

**【案例 4-1】 MR 英语单词频次统计**

原始数据：

```
The ASF provides an established framework
for intellectual property and financial
contributions that simultaneously limits
potential legal exposure for
our project committers
The ASF provides an established framework
for intellectual property and financial
contributions that simultaneously limits
potential legal exposure for
our project committers
```

输出结果：

```
ASF 2
The 2
an  2
and 2
committers  2
contributions   2
established 2
exposure    2
financial   2
for 4
framework   2
intellectual    2
legal   2
limits  2
our 2
potential   2
project 2
property    2
```

```
provides    2
simultaneously  2
that    2
```

原理：

首先对待处理的信息进行拆分，拆分之后在 map 阶段，把拆分的每个单词作为 map 方法的输出键，而 map 的方法输出的值设置为 1，最后在 reduce 阶段对每个键的值集合进行遍历并把遍历的值进行相加，输出结果即可。

（1）在 Linux 系统的命令终端上切换到/simple 目录，执行命令 touch source.txt，创建一个文件，如图 4-17 所示。

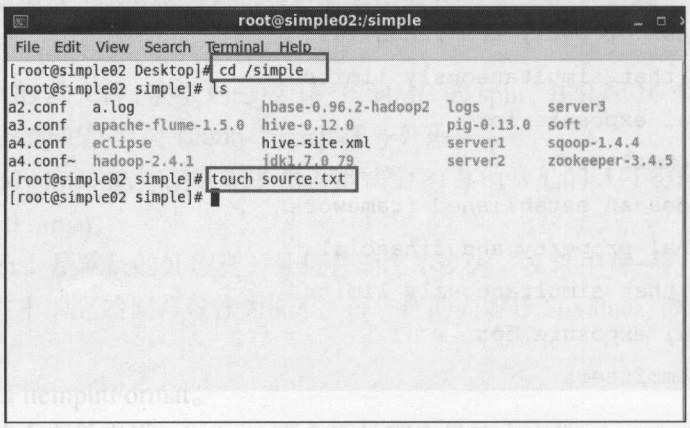



图 4-17　新建文件

（2）在 simple 目录下，执行命令 vi /simple/source.txt，编辑该文件，并把数据的信息内容拷贝到该文件中，然后在 simple 目录可以查看到 source.txt 文件，如图 4-18 所示。

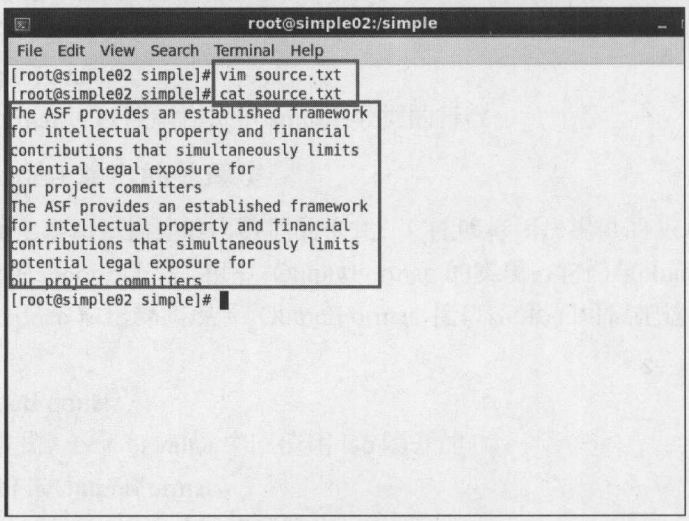



图 4-18　编译文件

（3）本案例如果在集群中需要用到 Hadoop 的存储和计算，所以在编写程序之前需要

先启动 Yarn 服务，可以在命令终端执行命令 start-all.sh， 把 HDFS 和 Yarn 进程启动，如图 4-19 所示。

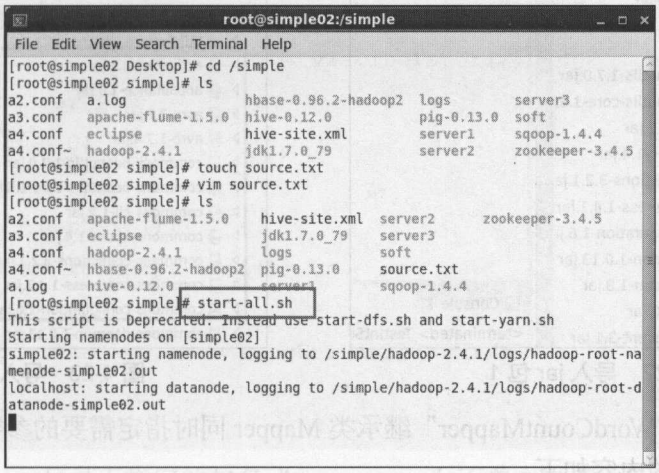



图 4-19 启动服务

（4）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”新建一个项目“EnglishWordsCount”，如图 4-20 所示。

（5）在项目 src 目录下，单击右键，选择“新建”，创建一个类文件名称为“WordCountMapper”并指定包名“com.wcount”，如图 4-21 所示。

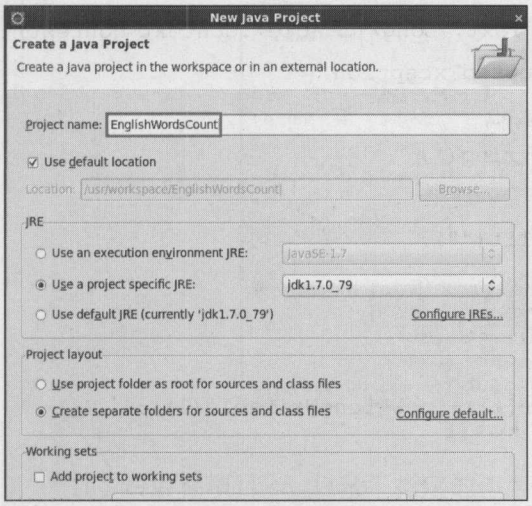



图 4-20 新建项目

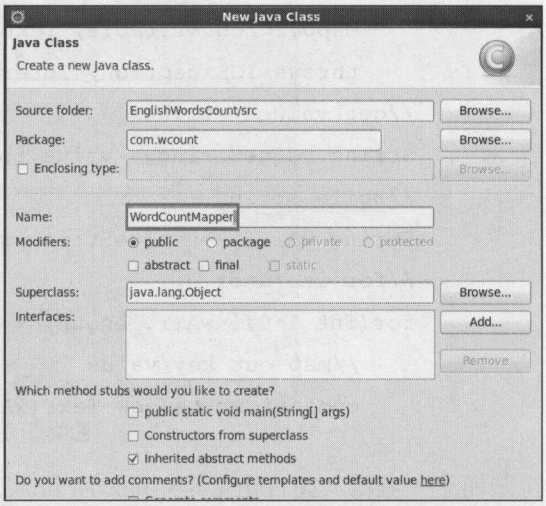



图 4-21 新建包名

（6）在编写“WordCountMapper”类之前需要把 Hadoop 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib 并把指定位置中的包放入该文件夹中，如图 4-22 所示。

（7）把 lib 下所有的 jar 包导入到环境变量，首先需要全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”，添加后，发现在项目下很多奶瓶图标的 jar 包，如图 4-23 所示。

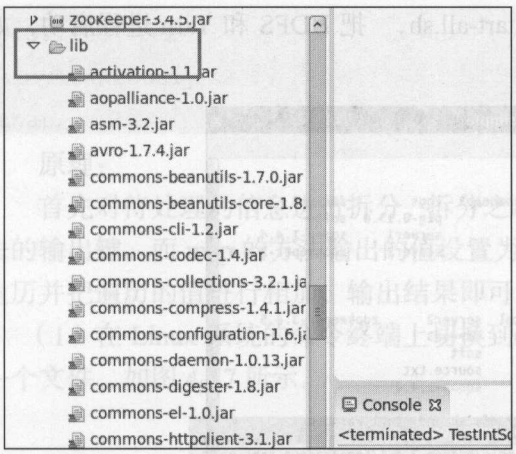



图 4-22　导入 jar 包 1

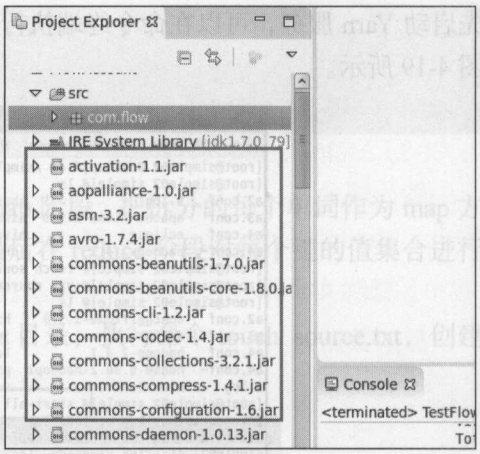



图 4-23　导入 jar 包 2

（8）编写类“WordCountMapper”继承类 Mapper 同时指定需要的参数类型，根据业务逻辑修改 map 类的内容如下。

```
package com.wcount;
public class WordCountMapper extends Mapper<LongWritable, Text, Text,
LongWritable>{
    @Override
    protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, Text, LongWritable>.Context context)
            throws IOException, InterruptedException {
        //get values string
        String valueString = value.toString();
        //spile string
        String wArr[] = valueString.split("");
        //for iterator
        for(int i=0;i<wArr.length;i++){
            //map out key/value
            context.write(new Text(wArr[i]), new LongWritable(1));
        }
    }
}
```

（9）在项目 src 目录下指定的包名“com.wcount”下单击右键，新建一个类名为“WordCountReducer ”并继承 Reducer 类，然后添加该类中的代码内容如下所示。

```
package com.wcount;
public class WordCountReducer extends Reducer<Text, LongWritable, Text,
LongWritable> {
    @Override
```

```
    protected void reduce(Text key, Iterable<LongWritable> v2s,
        Reducer<Text, LongWritable, Text, LongWritable>.Context context)
            throws IOException, InterruptedException {
        Iterator<LongWritable> it = v2s.iterator();
        //define var sum
        long sum = 0;
        // iterator count arr
        while(it.hasNext()){
            sum += it.next().get();
        }
        context.write(key, new LongWritable(sum));
    }
}
```

（10）在项目 src 目录下指定的包名“com.wcount”下单击右键，新建一个测试主类名为“TestMapReducer”并指定 main 主方法，如图 4-24 所示。

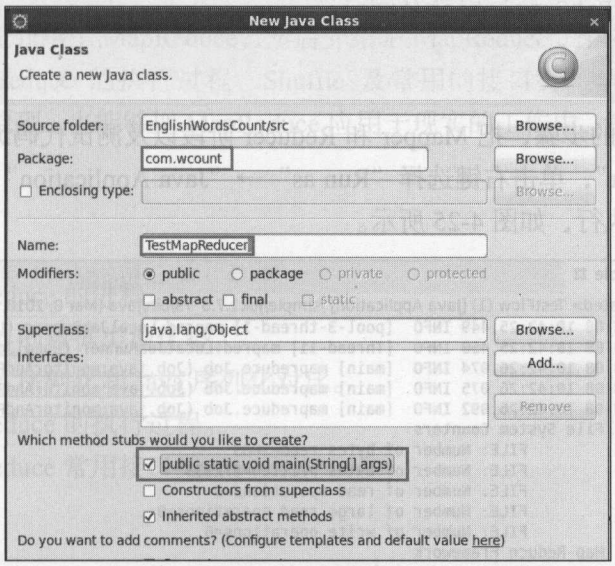



图 4-24 新建类名

（11）最后在项目 src 目录下指定的包名“com.wcount”下单击右键，新建一个测试主类名为“TestMapReducer”，添加测试代码如下所示。

```
package com.wcount;
public class TestMapReducer {
    public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        //conf.set("fs.default.name", "hdfs://192.168.0.202:9000");
        // step1 : get a job
```

```
        Job job = Job.getInstance(conf);
        //step2: set jar main class
        job.setJarByClass(TestMapReducer.class);
        //step3: set map class and redcer class
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        //step4: set map reduce output type
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        //step5: set key/value output file format and input/output path
FileInputFormat.setInputPaths(job,new Path("file:///simple/words.txt"));
FileOutputFormat.setOutputPath(job,new Path("file:///simple/wcout4"));
        //step6: commit job
        job.waitForCompletion(true);
    }
}
```

（12）按照以上的步骤，把 Mapper 和 Reducer 阶段以及测试代码编写完毕，选中测试类“TestMapReducer”，单击右键选择“Run as”→“Java Application”,查看控制台显示内容，查看是否正确执行，如图 4-25 所示。

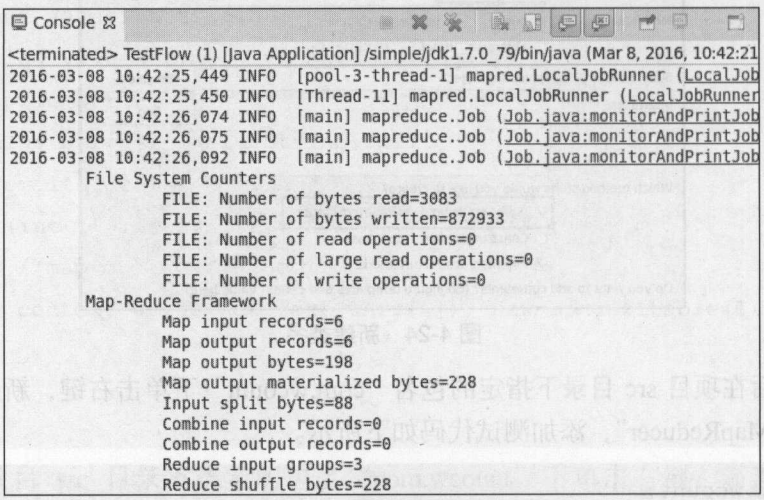



图 4-25　控制台查看信息

（13）程序执行完毕之后，可以到输出信息目录/simple/output 下，执行查看命令 cat part-r-00000 查看对数据处理后产生的结果，如图 4-26 所示。

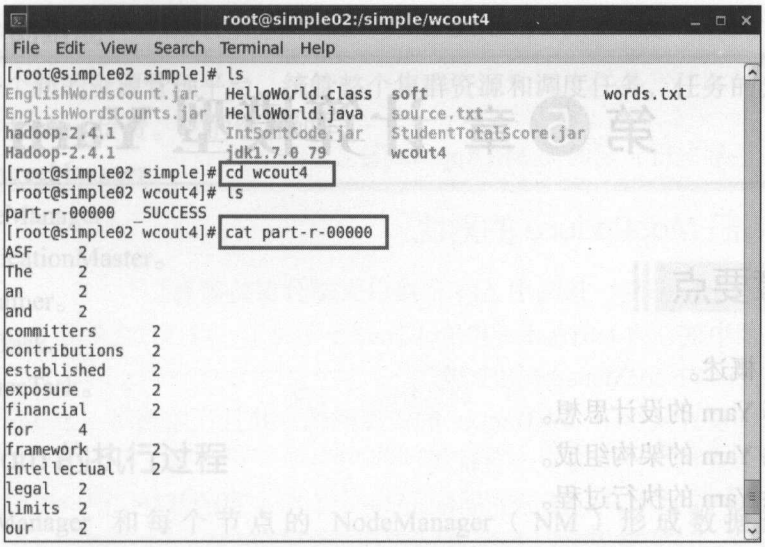



图 4-26 查看文件

## 本章小结

在本章中，首先介绍了 MapReduce，然后介绍了 MapReduce 的执行，例如数据分片、map 的执行过程、reduce 的执行过程、Shuffle 及常用的接口类。学习者不但能够理解 MapReduce 系统的原理，也能够把 MapReduce 应用于现实的工作中，解决海量数据计算的问题。

## 习题

1. 简述 MapReduce 的进程。
2. 简述 Hadoop 的数据类型优势。
3. 理解 Hadoop 序列化和 Java 序列化对比。
4. 简述 MapReduce 的执行过程。
5. 列举 MapReduce 常用接口类。

扫一扫在线测

# 第5章 计算模型 Yarn

## 本章要点

- Yarn 概述。
- 理解 Yarn 的设计思想。
- 熟悉 Yarn 的架构组成。
- 熟悉 Yarn 的执行过程。

## 引　言

Yarn 是由早期的 MapReduce 经过架构变换而来，主要思路是把原有 MapReduce 架构分为资源管理和监控调度两个部分。Yarn 舍弃了 MRv1 中的 JobTrack 和 TaskTrack，采用一种新的 MRAppMaster 进行管理，并与 Yarn 中的两个守护进程 ResourceManager 和 NodeManager 一起协同调度和控制任务，避免单一进程服务的管理和调度负载过重。本章通过对 Yarn 的概念介绍、Yarn 的执行过程、Yarn 和 MapReduce 的对比，让学生深刻理解和运用 Yarn。

## 5.1 Yarn 概述

### 5.1.1 Yarn 简介

MapReduce 在 0.23 的版本经过了一系列的优化，现在把 MapReduce 称为 MapReduce2.0（MRv2）或叫 Yarn，它能够支持多种编程模型，如 Storm、Spark。

Yarn 舍弃了 MRv1 中的 JobTrack 和 TaskTrack，采用一种新的 MRAppMaster 进行管理，并与 Yarn 中的两个守护进程 ResourceManager 和 NodeManager 一起协同调度和控制任务，避免单一进程服务的管理和调度负载过重。

MRv2 的原理是把 JobTracker 分成两部分，即资源管理和工作任务两个进程。也就是说有一个全局的进程 ResourceManager（RM）和每个应用有一个 NodeManager 进程。ResourceManager 和每个节点是主从关系，它在整个系统中，调配所有应用的资源。ResourceManager 接收到客户端任务请求后，会交给某个 NodeManager 并启动一个进程 MRAppMaster 负责任务的完成。MRAppMaster 分配任务到其他 DataNode 节点并启动 TaskMaster 进程。

### 5.1.2 Yarn 的组成

Yarn 是一个新的资源管理平台，统管整个集群资源和调度任务，任务的分配、运行和资源的配置都由 Yarn 负责。它的组成如下：

（1）ResourceManager。

（2）NodeManager。

（3）ApplicationMaster。

（4）Container。

（5）MapTask。

（6）ReduceTask。

## 5.2 Yarn 的执行过程

ResourceManager 和每个节点的 NodeManager（NM）形成数据计算框架。ResourceManager 是在整个框架系统中负责仲裁应用程序之间资源的最终决定者。

每个应用程序的 ApplicationMaster 实际上是一个框架特定的库，其任务是从 ResourceManager 协商资源以及对节点管理器（多个）的工作来执行和监视任务。

ResourceManager 中有两个主要组件：调度和 ApplicationsManager。

ApplicationMaster：检测状态和从调度中获取资源。

NodeManager：管理容器，检测资源使用情况并和 ResourceManager 报告信息。

Scheduler：负责分配资源给正在运行的应用程序。

从上面的描述中，可以看出 Yarn 的架构组成主要有 Client、ResourceManager、NodeManager 等进程，它们之间的工作协调过程如图 5-1 所示。

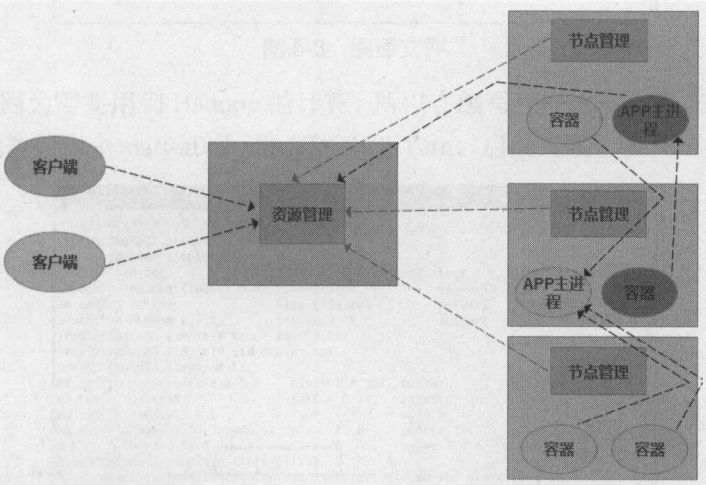



图 5-1 Yarn 计算过程

（1）Client 向 ResourceManager 提交任务。

（2）ResourceManager 分配创建 Container 任务并告知 NodeManager 启动进程 MRAppMaster。

（3）NodeManager 接收指定任务并开辟空间启动 MRAppMaster。

（4）NodeManager 完成任务之后会及时汇报给 ResourceManager。

（5）MRAppMaster 和 ResourceManager 交互，获取运行任务所需要的资源。

（6）MRAppMaster 获取资源后和 NodeManager 进行通信，启动 MapTask 或 ReduceTask 进程。

（7）任务正常运行，定时向 MRAppMaster 汇报工作情况。

## 5.3 新旧 MapReduce 的对比

（1）首先客户端不变，其调用 API 及接口大部分保持兼容。

（2）原框架中核心的 JobTracker 和 TaskTracker 不见了，取而代之的是 ResourceManager、ApplicationMaster 与 NodeManager 三个部分。

（3）新框架设计减少了 JobTracker 的资源消耗，并且让监测每一个 Job 子任务(tasks)状态的程序分布式化了，更安全、更优美。

（4）在新的 Yarn 中，ApplicationMaster 是一个可变更的部分，用户可以对不同的编程模型写自己的 AppMst，让更多类型的编程模型能够运行在 Hadoop 集群中。

（5）对于资源的表示以内存为单位，比之前以剩余 slot 数目更合理。

（6）老的框架中，JobTracker 一个很大的负担就是监控 job 下的 tasks 的运行状况，现在，这个部分就扔给 ApplicationMaster 做了，而 ResourceManager 中有一个模块叫作 ApplicationsMasters( 注意不是 ApplicationMaster)，它是监测 ApplicationMaster 的运行状况，如果出问题，会将其在其他机器上重启。

（7）Container 是 Yarn 为了将来做资源隔离而提出的一个框架。

**【案例 5-1】MR 计算整数的最大值和最小值**

原始数据：

```
102
10
39
109
200
11
3
90
28
5
2
30
838
10005
```

（1）在 Linux 系统的命令终端上切换到/simple 目录，执行命令 touch source.txt，创建一个文件，如图 5-2 所示。

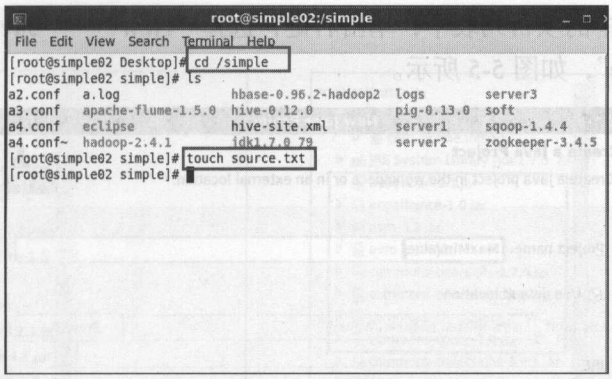



图 5-2 新建文件

（2）在 simple 目录下，执行命令 vi /simple/source.txt，编辑该文件，并把数据的信息内容拷贝到该文件中，然后在 simple 目录可以查看到 source.txt 文件，如图 5-3 所示。

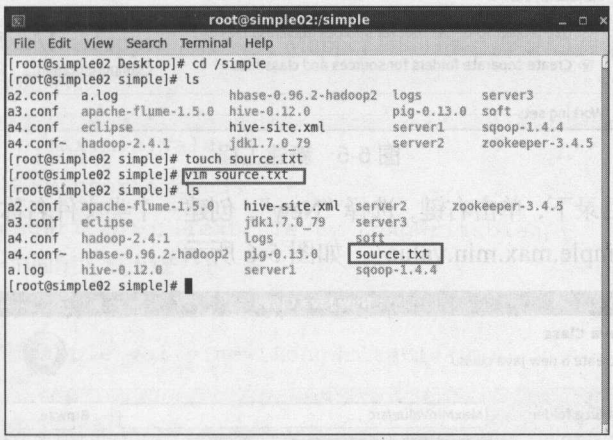



图 5-3 编译文件

（3）本案例因为需要用到 Hadoop 的计算，所以在编写程序之前需要先启动 Yarn 进程，可以在命令终端执行命令 start-all.sh 把 HDFS 和 Yarn，启动进程，如图 5-4 所示。

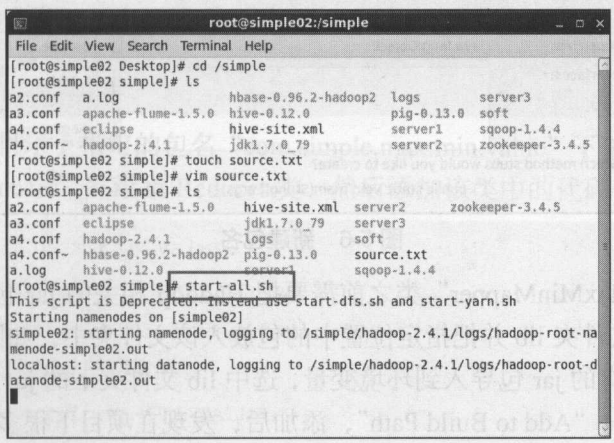



图 5-4 启动服务

（4）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”新建一个项目“MaxMinValue”，如图 5-5 所示。

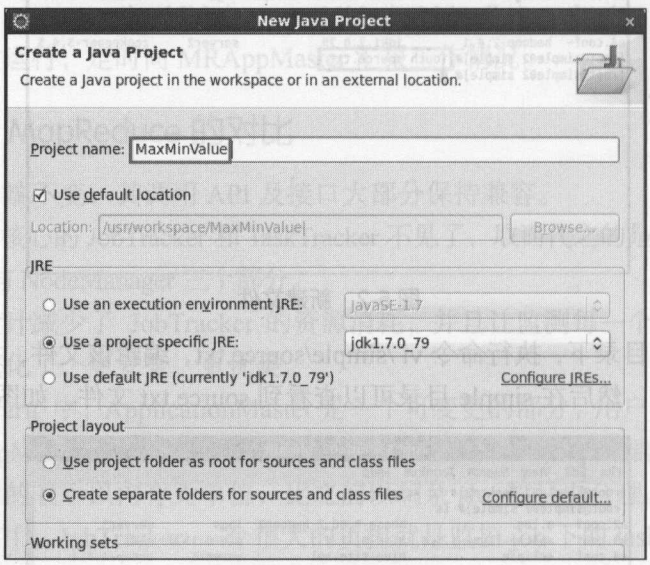



图 5-5　新建工程

（5）在项目 src 目录下，单击右键，选择“New”，创建一个类文件名称为“MaxMinMapper”并指定包名“com.simple.max.min.value”，如图 5-6 所示。

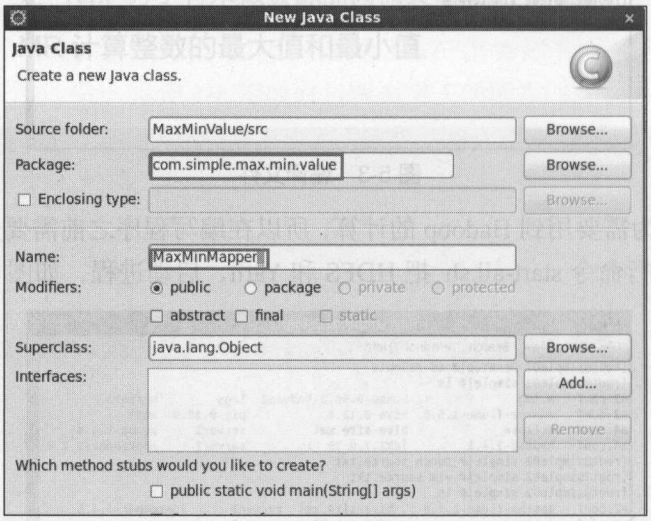



图 5-6　新建包名

（6）在编写“MaxMinMapper”类之前需要把 Hadoop 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib 并把指定位置中的包放入该文件夹中，如图 5-7 所示。

（7）把 lib 下所有的 jar 包导入到环境变量，选中 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”，添加后，发现在项目下很多奶瓶图标的 jar 包，如图 5-8 所示。

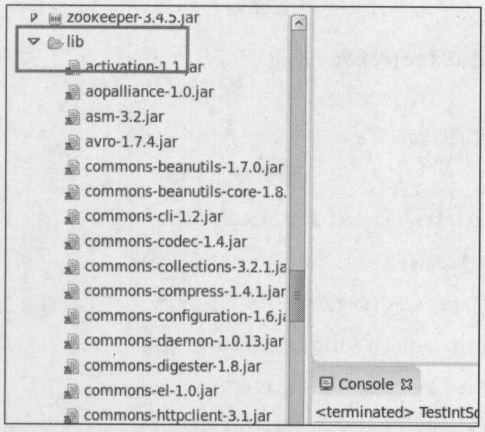



图 5-7 导入 jar 包

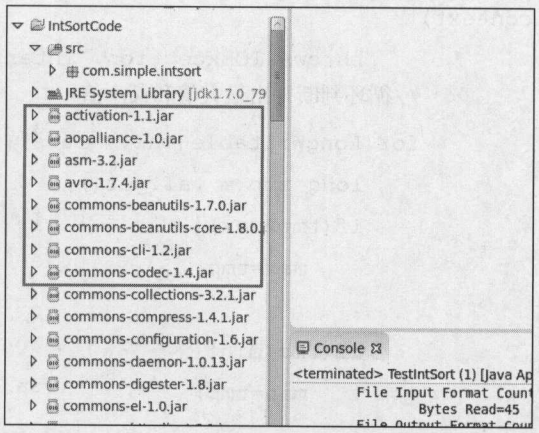



图 5-8 添加 jar 包

（8）编写类“MaxMinMapper”继承类 Mapper 同时指定需要的参数类型，根据业务逻辑修改 map 类的内容如下。

```
package com.simple.max.min.value;
public class MaxMinMapper extends
        Mapper<LongWritable, Text, Text, LongWritable> {
    //为 map 设置一个固定的输出键
    private final Text keyText = new Text("K");
    private LongWritable val = new LongWritable(0);
    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        //为 val 对象设置读取的数值
        val.set(Long.parseLong(value.toString()));
        context.write(keyText, val);
    }
}
```

（9）在项目 src 目录下指定的包名“com.simple.max.min.value”下单击右键，新建一个类名为“MaxMinReducer”并继承 Reducer 类，然后添加该类中的代码内容如下所示。

```
package com.simple.max.min.value;
public class MaxMinReducer extends
Reducer<Text, LongWritable, Text, Text> {
    //设置变量 max 和 min 的初始值
    private long max=Long.MIN_VALUE;
    private long min=Long.MAX_VALUE;
    @Override
```

```
    protected void reduce(Text key, Iterable<LongWritable> values,Context
context)
            throws IOException, InterruptedException {
        //循环判断得出最大值和最小值
        for(LongWritable val:values){
            long tmp = val.get();
            if(tmp>max){
                max=tmp;
            }
            if(tmp<min){
                min=tmp;
            }
        }
        //最大值和最小值分别输出
        context.write(new Text("Max"),new Text(max+"") );
        context.write(new Text("Min"), new Text(min+""));
    }
}
```

（10）在项目 src 目录下指定的包名“com.simple.max.min.value”下单击右键，新建一个测试主类名为“MaxMinJob”并指定 main 主方法，如图 5-9 所示。

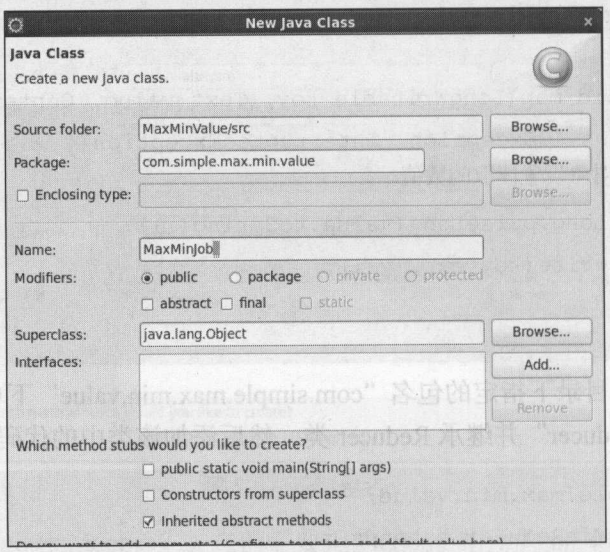



图 5-9　新建类名

（11）最后在项目 src 目录下指定的包名“com.simple.max.min.value”下单击右键，新建一个测试主类名为“MaxMinJob”，添加测试代码如下所示。

```
package com.simple.max.min.value;
public class MaxMinJob {
```

```
    public static void main(String[] args) throws Exception{
        //获取作业对象
        Job job = Job.getInstance(new Configuration());
        //设置主类
        job.setJarByClass(MaxMinJob.class);
        //设置 job 参数
        job.setMapperClass(MaxMinMapper.class);
        job.setReducerClass(MaxMinReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        //设置 job 输入输出
FileInputFormat.setInputPaths(job, new Path("file:///simple/source.txt"));
FileOutputFormat.setOutputPath(job, new Path("file:///simple/output"));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

（12）按照以上的步骤，把 Mapper 和 Reducer 阶段以及测试代码编写完毕之后，选中测试类“MaxMinJob”，单击右键选择“Run as”→“Java Application”，查看控制台显示内容，查看是否正确执行，如图 5-10 所示。

（13）程序执行完毕之后，可以到输出信息目录/simple/output 下，执行查看命令 cat part-r-00000 查看对数据处理后产生的结果，如图 5-11 所示。

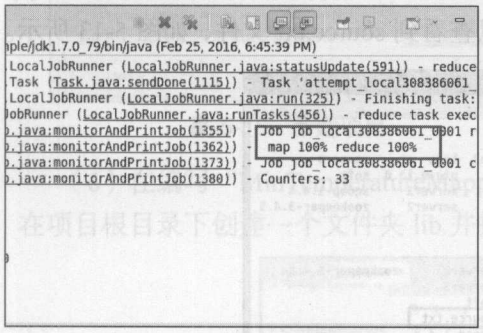



图 5-10 控制台信息

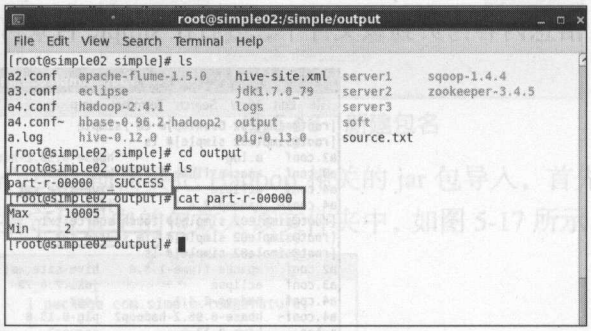



图 5-11 查看文件信息

## 【案例 5-2】MapReduce 分析年气象数据最低温度

把采集的气象数据信息以日志的方式保存到指定的位置，该位置可以是本地，也可以是 HDFS 分布式系统上。利用 Hadoop 计算技术对该日志文件进行处理，主要分两个阶段：Mapper 阶段和 Reducer 阶段。Mapper 阶段主要是对日志文件进行按行读取并进行字符串截取，Reducer 阶段对 Mapper 阶段传过来的数据进行大小比较，最终获取每一年中的最高温度。

气象采集数据格式如下：

0067011990999991950051507004888888889999999N9+00001+99999999999999999999

0067011990999991950051512004888888889999999N9+00221+99999999999999999999

0067011990999991950051518004888888889999999N9-00111+99999999999999999999

0067011990999991949032412004888888889999999N9+01111+99999999999999999999

数据说明：

第 15 ~ 19 个字符是 year。

第 45 ~ 50 位是温度表示，+表示零上，-表示零下，且温度的值不能是 9999，9999 表示异常数据。

第 50 位值只能是 0、1、4、5、9 几个数字。

（1）在 Linux 系统的命令终端上切换到/simple 目录，执行命令 touch source.txt 创建一个文件，如图 5-12 所示。

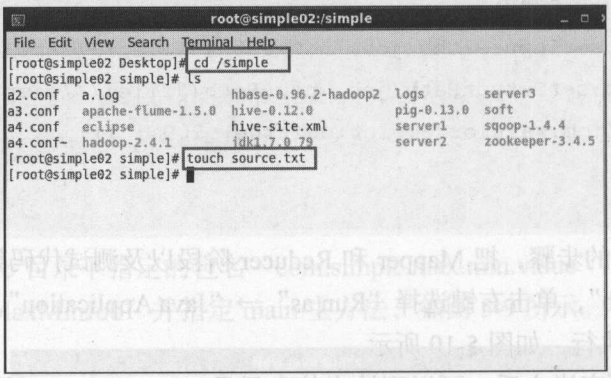



图 5-12　新建文件

（2）在 simple 目录下，执行命令 vim /simple/source.txt，编辑该文件，并把采集的气象信息内容拷贝到该文件中，然后在 simple 目录可以查看到 source.txt 文件，如图 5-13 所示。

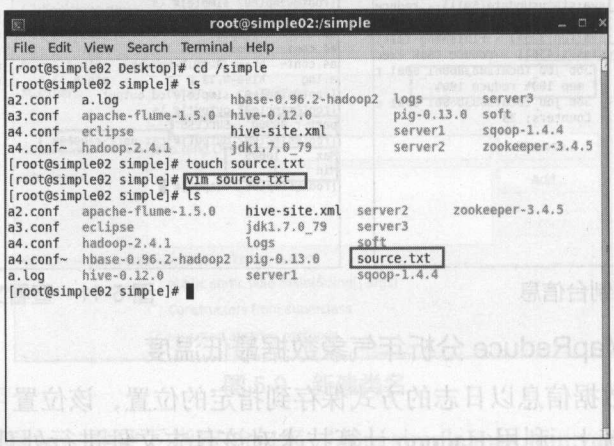



图 5-13　编译文件

（3）本案例因为需要用到 Hadoop 的计算，所以在编写程序之前需要先启动 yarn 进程，可以在命令终端执行命令 start-all.sh 启动 HDFS 和 Yarn 进程，如图 5-14 所示。

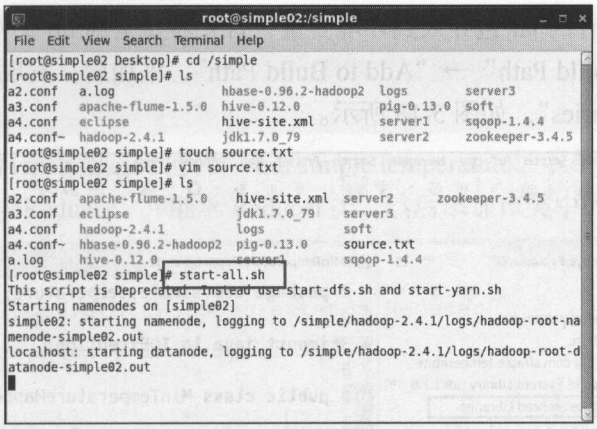



图 5-14 启动服务

（4）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”，新建一个项目“TemperatureLow”，如图 5-15 所示。

（5）在项目 src 目录下，单击右键，选择“New”创建一个类文件名称为“MinTemperature Mapper”并指定包名“com.simple.temperature”，如图 5-16 所示。

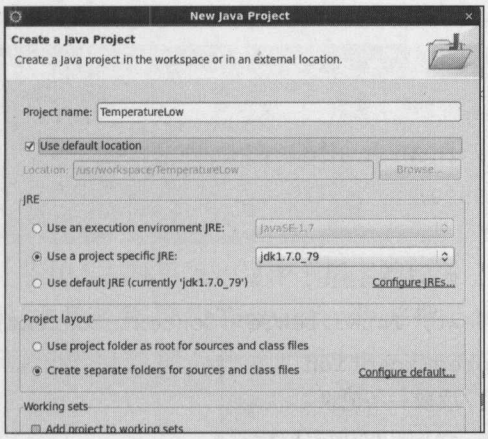



图 5-15 新建项目

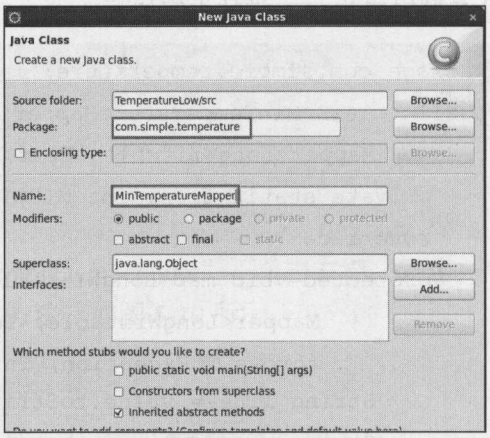



图 5-16 新建包名

（6）在编写“MinTemperatureMapper”类之前需要把 Hadoop 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib 并把指定位置中的包放入该文件夹中，如图 5-17 所示。

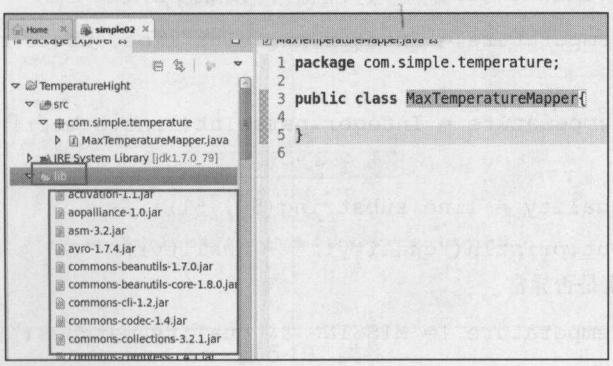



图 5-17 导入 jar 包

（7）把 lib 下所有的 jar 包导入到环境变量，首先需要全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”，添加后，发现在项目下多一个列表项“Referenced Libraries”，如图 5-18 所示。

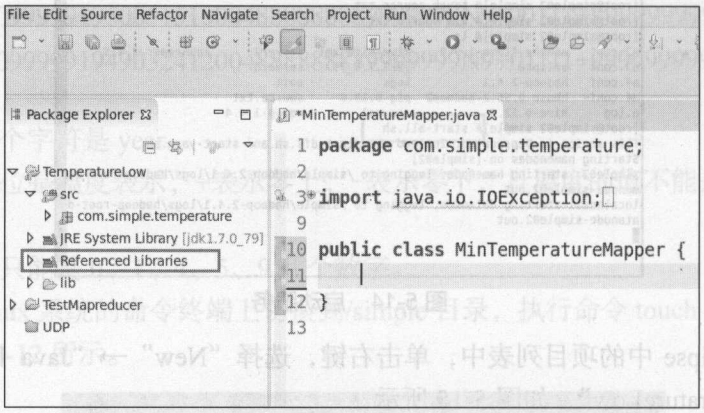



图 5-18　添加 jar 包

（8）编写类“MinTemperatureMapper”继承类 Mapper 同时指定需要的参数类型，根据业务逻辑修改 map 类的内容如下。

```
package com.simple.temperature;
public class MinTemperatureMapper extends
        Mapper<LongWritable, Text, Text, IntWritable> {
    private static final int MISSING = 9999;
    @Override
    protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, Text, IntWritable>.Context context)
            throws IOException, InterruptedException {
        String line = value.toString();//读取一条记录
        String year = line.substring(15, 19);//获取温度数
        System.out.println("year=="+year);
        int airTemperature;
        if (line.charAt(45) == '+') { //判断温度正负
            airTemperature = Integer.parseInt(line.substring(46, 50));
        } else {
            airTemperature = Integer.parseInt(line.substring(45, 50));
        }
        String quality = line.substring(50, 51);
        System.out.println("quality: " + quality);
        //判断温度是否异常
        if (airTemperature != MISSING && quality.matches("[01459]")) {
            context.write(new Text(year), new IntWritable(airTemperature));
```

```
        }
    }
}
```

（9）在项目 src 目录下指定的包名“com.simple.temperature”下单击右键，新建一个类名为“MinTemperatureReducer”并继承 Reducer 类，然后添加该类中的代码内容如下所示。

```
package com.simple.temperature;
public class MinTemperatureReducer extends Reducer
<Text, IntWritable, Text, IntWritable>{
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
            Context context)
            throws IOException, InterruptedException {
        //把 Integer.Max_VALUE 作为 minValue 的初始值
        int minValue = Integer.MAX_VALUE;
        //循环取出最大值
        for(IntWritable value: values){
            maxValue = Math.max(minValue, value.get());
        }
        context.write(key, new IntWritable(minValue));
    }
}
```

（10）在项目 src 目录下指定的包名“com.simple.temperature”下单击右键，新建一个测试主类名为“MinTemperature”并指定 main 主方法，如图 5-19 所示。

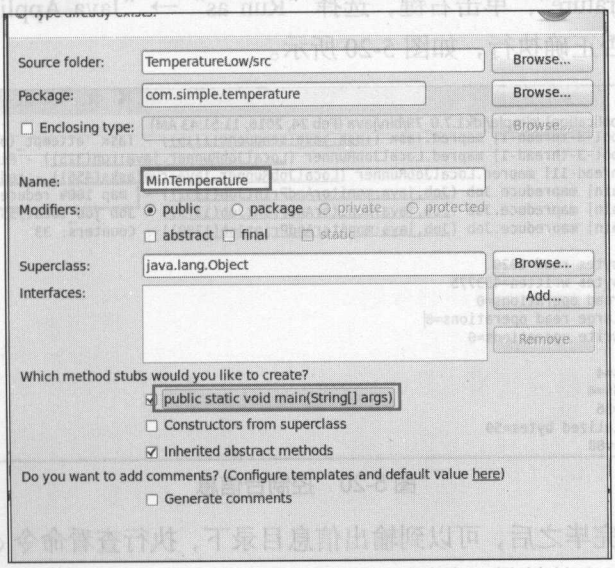



图 5-19 类名

（11）最后在项目 src 目录下指定的包名“com.simple.temperature”下单击右键，新建一个测试主类名为“MinTemperature”，添加测试代码如下所示。

```
package com.simple.temperature;
public class MinTemperature {
    public static void main(String[] args) throws Exception{
        //获取作业对象
        Job job = Job.getInstance(new Configuration());
        //设置主类
        job.setJarByClass(MinTemperature.class);
        //设置 job 参数
        job.setMapperClass(MinTemperatureMapper.class);
        job.setReducerClass(MinTemperatureReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        //设置 job 输入输出
        FileInputFormat.addInputPath(job,
            new Path("file:///simple/source.txt"));
        FileOutputFormat.setOutputPath(job,
            new Path("file:///simple/output"));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

（12）按照以上的步骤，把 Mapper 和 Reducer 阶段以及测试代码编写完毕之后，选中测试类“MinTemperature”，单击右键，选择“Run as”→“Java Application”,查看控制台显示内容，查看是否正确执行，如图 5-20 所示。

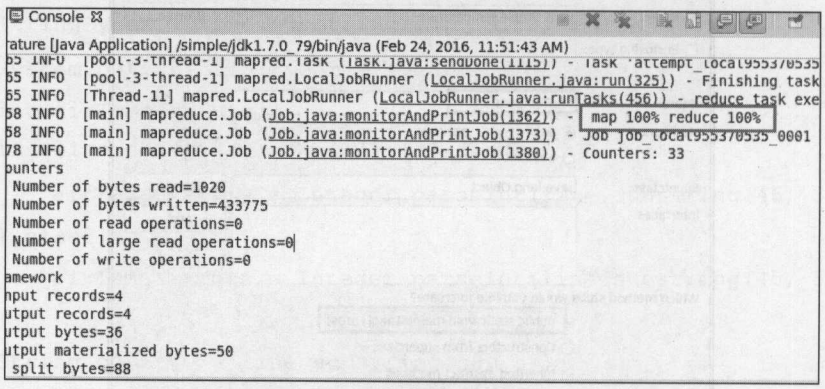



图 5-20　控制台信息

（13）程序执行完毕之后，可以到输出信息目录下，执行查看命令 cat part-r-00000 查看对采集数据处理后产生的结果，如图 5-21 所示。

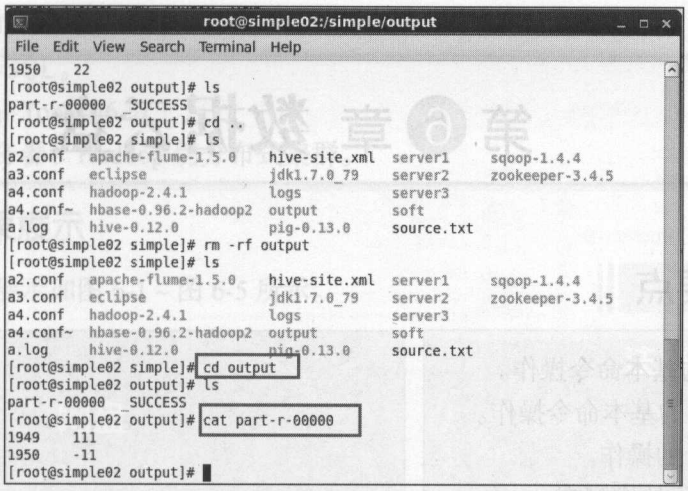



图 5-21 查看文件信息

## 本章小结

在本章中，首先介绍了 Yarn 的概念，然后介绍 Yarn 的架构组成，最后介绍了 Yarn 的执行。学习者不但能够理解 Yarn 系统的原理，也能够把 Yarn 应用于现实的工作当中，解决海量数据计算的问题。

## 习题

1. 简述 Yarn 架构的进程。
2. 简述 Hadoop 的数据类型优势。
3. 理解 Yarn 和 MapReduce 的对比。

扫一扫在线测

# 第6章 数据云盘

## 本章要点

- Linux 的基本命令操作。
- Hadoop 的基本命令操作。
- MySQL 的操作。
- Web 项目的增删查。

## 引　言

本章就是通过介绍如何开发一个基于 Hadoop 实现的网盘，来理解开发的原理及相关的技术知识点。让读者在实践中学会 Hadoop、HDFS 的操作。

## 6.1 项目概述

网盘是基于云计算理念推出的企业数据网络存储和管理解决方案，利用互联网后台数据中心的海量计算和存储能力为企业提供数据汇总分发、存储备份和管理等服务。基于 Hadoop 开发这样一个系统，主要包括系统前台、系统后台两个部分。本文主要讲解了系统后台的开发流程。通过采用 JavaEE 技术和云存储技术对系统进行设计以及具体实现。系统后台的开发是采用 Java 语言，使用 Spring、bootmetro-master 等 JavaEE 开发框架，使用 Hadoop 的伪分布式文件系统存储文件。

## 6.2 功能需求

登录:用户登录个人账户。

注册:用户注册账户。

上传:从用户本地发送文件到服务器。

查询:查看服务器上存储文件目录。

下载:用户向服务器发送请求资源的信息，服务器根据资源信息发送相应文件到用户本地。

删除:用户操作删除服务器上的资源。

## 6.3 软件开发需求

开发工具：Eclipse。

开发语言：Java。
数据库：MySQL。
应用服务器：Tomcat6 以上。
数据存储服务器：Hadoop 伪分布式集群。

## 6.4 效果展示

网盘的效果展示如图 6-1 ~ 图 6-5 所示。

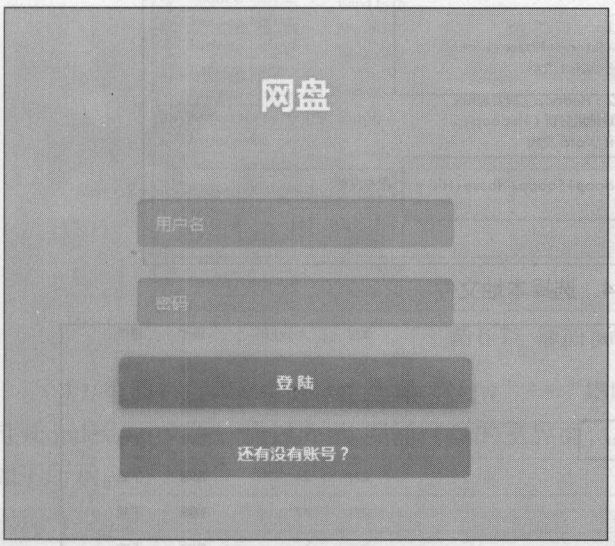



图 6-1 登录界面

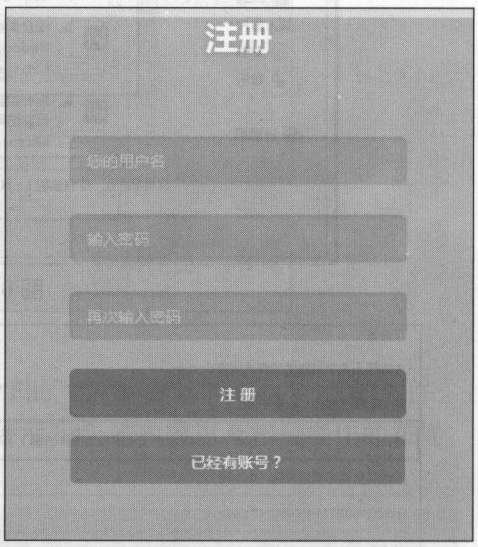



图 6-2 注册界面

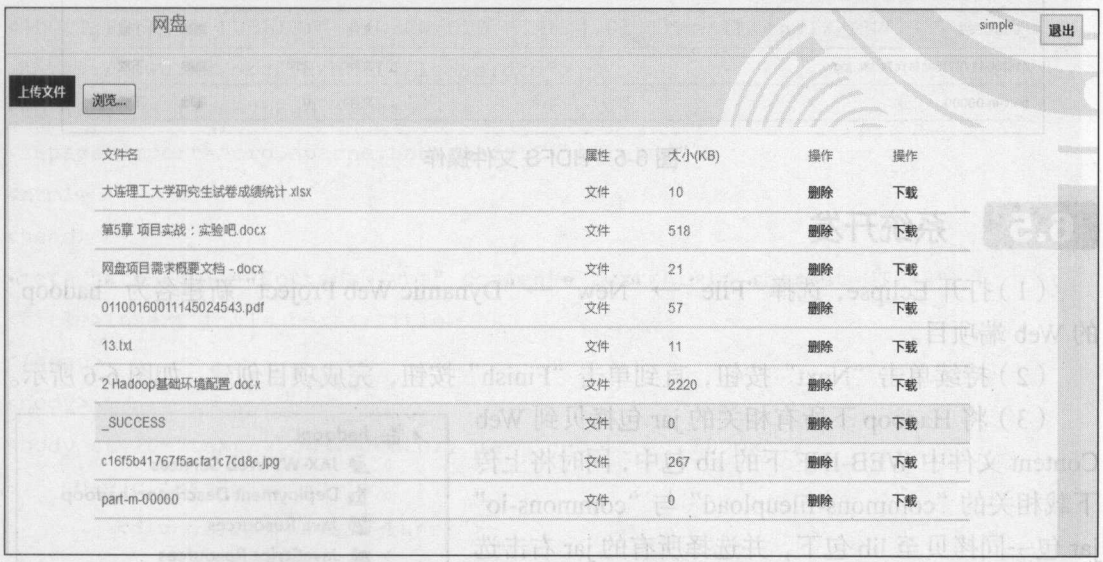



图 6-3 HDFS 文件上传与列表展示


图 6-4　选择本地文件

| 文件名 | 属性 | 大小(KB) | 操作 | 操作 |
|---|---|---|---|---|
| 大连理工大学研究生试卷成绩统计.xlsx | 文件 | 10 | 删除 | 下载 |
| 第5章 项目实战：实验吧.docx | 文件 | 518 | 删除 | 下载 |
| 综合实验1：Hadoop+Sqoop+Hbase+Hive（环境搭建操作步骤）.docx | 文件 | 965 | 删除 | 下载 |
| 网盘项目需求概要文档 -.docx | 文件 | 21 | 删除 | 下载 |
| 01100160011145024543.pdf | 文件 | 57 | 删除 | 下载 |
| 13.txt | 文件 | 11 | 删除 | 下载 |
| 2 Hadoop基础环境配置.docx | 文件 | 2220 | 删除 | 下载 |
| _SUCCESS | 文件 | 0 | 删除 | 下载 |
| c16f5b41767f5acfa1c7cd8c.jpg | 文件 | 267 | 删除 | 下载 |
| part-m-00000 | 文件 | 0 | 删除 | 下载 |

图 6-5　HDFS 文件操作

## 6.5　系统开发

（1）打开 Eclipse，选择“File”→“New”→“Dynamic Web Project”新建名为“hadoop”的 Web 端项目。

（2）持续单击“Next”按钮，直到单击“Finish”按钮，完成项目创建，如图 6-6 所示。

（3）将 Hadoop 下所有相关的 jar 包拷贝到 Web Content 文件中 WEB-INF 下的 lib 包中，同时将上传下载相关的“commons-fileupload”与“commons-io”jar 包一同拷贝至 lib 包下，并选择所有的 jar 右击选择“Build Path”→“Add to Build Path”在项目中添加需要的 jar 包。如图 6-7 所示。

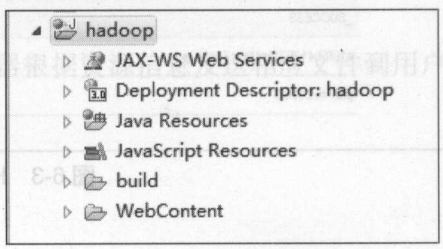



图 6-6　项目创建

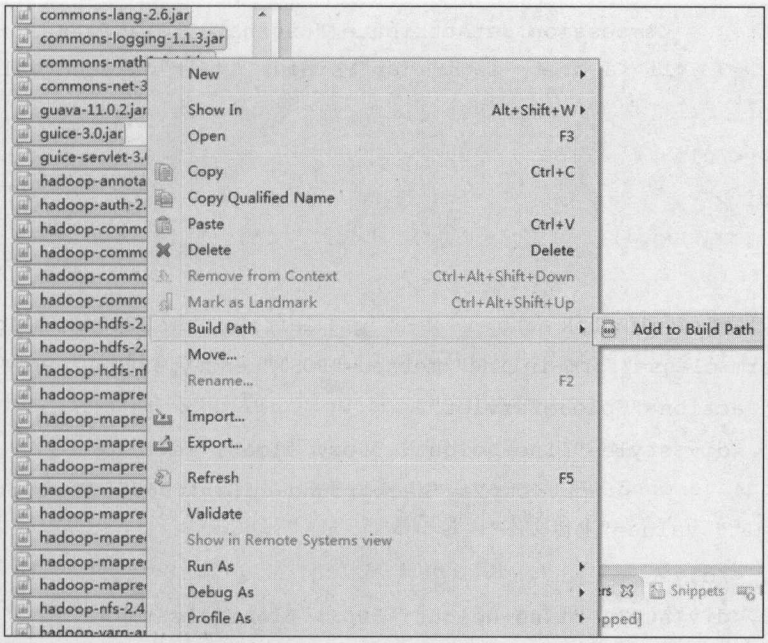



图 6-7 添加 jar

（4）单击右键 WebContent 选择“New”→“JSP File”新建名为“index.jsp”的文件，打开 index.jsp 文件，通过 html 编译网盘的主界面，展示 hdfs 的内容以及相关操作的布局，如下所示。

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<%@ include file="head.jsp"%>
<%@page import="org.apache.hadoop.fs.FileStatus"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<body style="text-align: center; margin-bottom: 100px;">
    <div class="navbar">
        <div class="navbar-inner">
            <a class="brand" href="#" style="margin-left: 200px;"
 style="color:#000000">网盘</a>
            <ul class="nav" style="line-height: 50px; float: right;">
                <li><a style="color:#000000 size=40">
```

```
                <%=session.getAttribute("username") %></a></li>
                <li><a href="login.jsp"><input type="button" value="退出
"></a></li>
            </ul>
        </div>
    </div>
    <div
        style="margin: 0px auto; text-align: left; width: 1200px; height: 50px;">
        <form class="form-inline" method="POST" enctype="MULTIPART/FORM-DATA"
            action="UploadServlet">
            <div style="line-height: 50px; float: left;">
                    <input style="background-color:#000000"  type="submit"
 name="submit" value="上传文件">
            </div>
            <div style="line-height: 50px; float: left;">
                <input type="file" name="file1" size="30"/>
            </div>
        </form>
    </div>
    <div
        style="margin: 0px auto; width: 1200px; height: 500px; background: #fff">
        <table class="table table-hover"
            style="width: 1000px; margin-left: 100px;">
            <tr>
                <td>文件名</td>
                <td>属性</td>
                <td>大小(KB)</td>
                <td>操作</td>
                <td>操作</td>
            </tr>
            <%
        FileStatus[] list = (FileStatus[])request.getAttribute("documentList");
                String name = (String)request.getAttribute("username");
            if(list != null)
            for (int i=0; i<list.length; i++) {
                %>
            <tr style="border-bottom: 2px solid #ddd">
                <%
                        if(list[i].isDir())//DocumentServlet
```

```
                    {
                        out.print("<td><a
href=\"UploadServlet?filePath="+list[i].getPath()+"\">"+list[i].getPath().
getName()+"</a></td>");
                    }else{
                out.print("<td>"+list[i].getPath().getName()+"</td>");
                    }
                %>
            <td><%= (list[i].isDir()?"目录":"文件") %></td>
            <td><%= list[i].getLen()/1024%></td>
            <td ><a style="color:#000000"
    href="DeleteFileServlet?filePath=<%=java.net.URLEncoder.encode(list[i]
.getPath().toString(),"GB2312") %>">删除</a></td>
            <td ><a style="color:#000000" c
    href="DownloadServlet?filePath=<%=java.net.URLEncoder.encode(list[i].g
etPath().toString(),"GB2312") %>">下载</a></td>
        </tr>
        <%
        }
        %>
    </table>
</div>
</body>
</body>
</html>
```

（5）单击右键 hadoop/Java Resources 下的 src，选择“New”→“Package”，新建名为“com.simple.controller”包名，用相同操作步骤，新建名为“com.simple.bean”与“com.simple.model”包名。

（6）右击 com.simple.controller 包名，选择“New”→“Class”新建名为“UploadServlet”的类文件并继承 HttpServlet ，重写 doGet 和 doPost 方法，对该类进行编译，操作对本地文件上传至 HDFS 的内容，基于安全考虑，无法获取文件的绝对路径，首先将得到的文件保存至本地指定的地址，再从指定的地址拼接文件的 url 得到文件的绝对路径，将文件上传至 hdfs 上，如下所示。

```
package com.simple.controller;
public class UploadServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        this.doPost(request, response);
```

```
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.setCharacterEncoding("UTF-8");
        File file ;
        int maxFileSize = 50 * 1024 *1024;  //50M
        int maxMemSize = 50 * 1024 *1024;    //50M
        ServletContext context = getServletContext();
        String filePath = context.getInitParameter("file-upload");
        System.out.println("source file path:"+filePath+"");
        // 验证上传内容的类型
        String contentType = request.getContentType();
        if ((contentType.indexOf("multipart/form-data") >= 0)) {
            DiskFileItemFactory factory = new DiskFileItemFactory();
            // 设置内存中存储文件的最大值
            factory.setSizeThreshold(maxMemSize);
            // 本地存储的数据大于 maxMemSize.
            factory.setRepository(new File("c:\\temp"));
            // 创建一个新的文件上传处理程序
            ServletFileUpload upload = new ServletFileUpload(factory);
            // 设置最大上传的文件大小
            upload.setSizeMax( maxFileSize );
            try{
                // 解析获取的文件
                List fileItems = upload.parseRequest(request);
                // 处理上传的文件
                Iterator i = fileItems.iterator();
                System.out.println("begin to upload file to tomcat server</p>");
                while ( i.hasNext () ) {
                    FileItem fi = (FileItem)i.next();
                    if ( !fi.isFormField () )  {
                        // 获取上传文件的参数
                        String fieldName = fi.getFieldName();
                        String fileName = fi.getName();
                        String fn =
 fileName.substring( fileName.lastIndexOf("\\")+1);
                        System.out.println("<br>"+fn+"<br>");
                        boolean isInMemory = fi.isInMemory();
                        long sizeInBytes = fi.getSize();
```

```
                    // 写入文件
                    if( fileName.lastIndexOf("\\") >= 0 ){
                        file = new File( filePath ,
                        fileName.substring( fileName.lastIndexOf("\\"))) ;
                    }else{
                        file = new File( filePath ,
                    fileName.substring(fileName.lastIndexOf("\\")+1)) ;
                    }
                    fi.write( file ) ;
                System.out.println("upload file to tomcat server success!");
                System.out.println("begin to upload file to hadoop hdfs</p>");
                    String name = filePath + "\\"+ fileName;
                    //将 tomcat 上的文件上传到 hadoop 上
                    HdfsDAO hdfs = new HdfsDAO();
                    hdfs.copyFile(name);
                System.out.println("upload file to hadoop hdfs success!");
                    FileStatus[] documentList = hdfs.getDirectoryFromHdfs();
                    request.setAttribute("documentList",documentList);
                    System.out.println("得到 list 数据"+documentList);
        request.getRequestDispatcher("index.jsp").forward(request, response);
                }
            }
        }catch(Exception ex) {
            System.out.println(ex);
        }
    }else{
        System.out.println("<p>No file uploaded</p>");
    }
  }
}
```

（7）单击右键 com.simple.model 包名选择“New”→“Class”新建名为“HdfsDAO”的类文件，操作对 HDFS 的增删改查的方法，需要对 hdfs 做什么操作，只需在本类中调用相应的方法，如下所示。

```
package com.simple.model;
public class HdfsDAO {
    private static String hdfsPath = "hdfs://192.168.65.133:9000/usr/simple/";
    Configuration conf = new Configuration();
    /**上传文件到 HDFS 上去*/
```

```
    public void copyFile(String local) throws IOException {
        FileSystem fs = FileSystem.get(URI.create(hdfsPath),conf);
        //remote---/用户/用户下的文件或文件夹
        fs.copyFromLocalFile(new Path(local), new Path(hdfsPath));
        fs.close();
    }
  /** 从 HDFS 上下载数据**/
  public void download(String remote, String local) throws IOException {
    FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);
    fs.copyToLocalFile(false,new Path(remote), new Path(local),true);
    System.out.println("download: from" + remote + " to " + local);
    fs.close();
  }
  /**从 HDFS 上删除文件*/
  public static void deleteFromHdfs(String deletePath) throws
FileNotFoundException,IOException {
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(URI.create(deletePath), conf);
      fs.deleteOnExit(new Path(deletePath));
      fs.close();
  }  /**遍历 HDFS 上的文件和目录*/
  public static FileStatus[] getDirectoryFromHdfs() throws
FileNotFoundException,IOException {
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);
      FileStatus[] list = fs.listStatus(new Path(hdfsPath));
    if(list != null)
    for (FileStatus f : list) {
        System.out.printf("name: %s, folder: %s, size: %d\n",
f.getPath().getName(), f.isDir(), f.getLen());
    }
    fs.close();
    return  list;
  }
}
```

（8）打开“WEB-INF”下的 web.xml 文件，为 UploadServlet 配置<servlet-class><url-pattern>相关的信息连接。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>hadoop</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
    <servlet>
        <description></description>
        <display-name>UploadServlet</display-name>
        <servlet-name>UploadServlet</servlet-name>
        <servlet-class>com.simple.controller.UploadServlet</servlet-class>
    </servlet>
    <context-param>
        <description>Location to store uploaded file</description>
        <param-name>file-upload</param-name>
        <param-value>
        D:\webapp\data\data
    </param-value>
    </context-param>
    <servlet-mapping>
        <servlet-name>UploadServlet</servlet-name>
        <url-pattern>/UploadServlet</url-pattern>
    </servlet-mapping>
</web-app>
```

（9）通过 start-all.sh 命令启动 Hadoop，通过 jps 查看 Hadoop 是否启动成功，通过浏览器访问启动 Hadoop，路径为本机的 IP，端口为 50070，出现如下界面，表示本机可以访问 Hadoop，如图 6-8 所示。

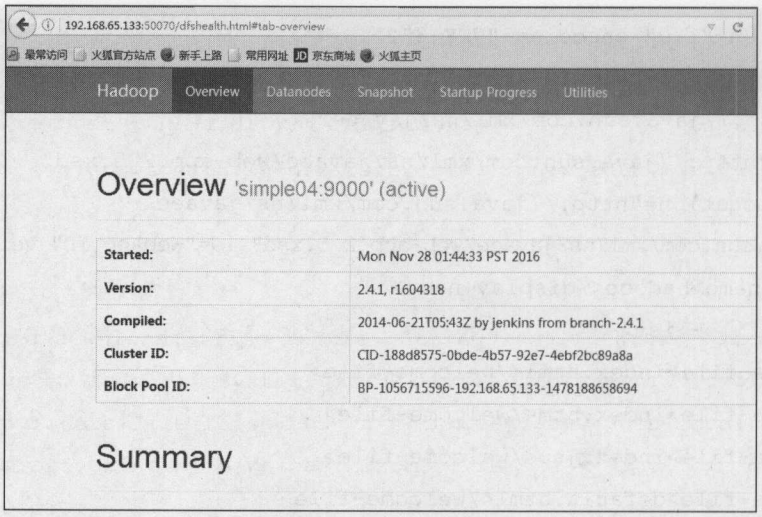



图 6-8　Hadoop 测试

（10）单击右键 Eclipse 中的 Hadoop 项目，选择“Run As”→“Run As Server”运行项目，在浏览器中输入 IP 访问该项目，出现如下界面，通过浏览本地文件，单击上传文件将文件上传至 HDFS 中，同时会在下方展示 HDFS 中的文件相关信息，如图 6-9 所示。

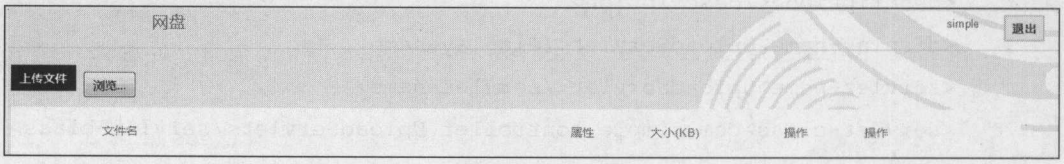



图 6-9　上传界面

（11）单击右键 com.simple.controller 包名，选择“New”→“Class”，新建名为“DeleteFileServlet”的类文件，操作对 HDFS 文件删除的内容。

```
package com.simple.controller;
public class DeleteFileServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String filePath = new
 String(request.getParameter("filePath").getBytes("ISO-8859-1"),"GB2312");
        HdfsDAO hdfs = new HdfsDAO();
        hdfs.deleteFromHdfs(filePath);
        System.out.println("====="+filePath+"=====");
        FileStatus[] documentList = hdfs.getDirectoryFromHdfs();
        request.setAttribute("documentList",documentList);
        System.out.println("得到 list 数据"+documentList);
        request.getRequestDispatcher("index.jsp").forward(request,response);
    }
```

```
    protected void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        this.doGet(request, response);
    }
}
```

（12）打开“WEB-INF”下的 web.xml 文件，添加如下内容，为 DeleteFileServlet 配置 <servlet-class> <url-pattern>相关的信息连接。

```
<servlet>
   <description></description>
   <display-name>DeleteFileServlet</display-name>
   <servlet-name>DeleteFileServlet</servlet-name>
   <servlet-class>com.simple.controller.DeleteFileServlet</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>DeleteFileServlet</servlet-name>
   <url-pattern>/DeleteFileServlet</url-pattern>
 </servlet-mapping>
```

（13）相同步骤运行项目并在浏览器中进行查看，在列表操作项中单击删除即可删除 HDFS 中相关的文件，如图 6-10 所示。

| 文件名 | 属性 | 大小(KB) | 操作 | 操作 |
| --- | --- | --- | --- | --- |
| 新建文本文档 (2).txt | 文件 | 2 | 删除 | 下载 |
| 第5章 项目实战：实验吧.docx | 文件 | 518 | 删除 | 下载 |
| 网盘项目需求概要文档 -.docx | 文件 | 21 | 删除 | 下载 |
| 01100160011145024543.pdf | 文件 | 57 | 删除 | 下载 |
| 13.txt | 文件 | 11 | 删除 | 下载 |
| 2 Hadoop基础环境配置.docx | 文件 | 2220 | 删除 | 下载 |
| _SUCCESS | 文件 | 0 | 删除 | 下载 |
| c16f5b41767f5acfa1c7cd8c.jpg | 文件 | 267 | 删除 | 下载 |
| part-m-00000 | 文件 | 0 | 删除 | 下载 |

图 6-10 删除操作

（14）右击 com.simple.controller 包名，选择“New”→“Class”新建名为“DownloadServlet”的类文件，操作对文件下载的内容。

```
package com.simple.controller;
public class DownloadServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        String local = "F:\\";
```

```
        String filePath = new
 String(request.getParameter("filePath").getBytes("ISO-8859-1"),"GB2312");
        HdfsDAO hdfs = new HdfsDAO();
        hdfs.download(filePath, local);
        FileStatus[] documentList = hdfs.getDirectoryFromHdfs();
        request.setAttribute("documentList",documentList);
        System.out.println("得到 list 数据"+documentList);
        request.getRequestDispatcher("index.jsp").forward(request,response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        this.doGet(request, response);
    }
}
```

（15）打开“WEB-INF”下的 web.xml 文件，添加如下内容，为 DownloadServlet 配置 <servlet-class> <url-pattern>相关的信息连接。

```
<servlet>
    <description></description>
    <display-name>DownloadServlet</display-name>
    <servlet-name>DownloadServlet</servlet-name>
    <servlet-class>com.simple.controller.DownloadServlet</servlet-class>
 </servlet>
 <servlet-mapping>
    <servlet-name>DownloadServlet</servlet-name>
    <url-pattern>/DownloadServlet</url-pattern>
 </servlet-mapping>
```

（16）相同步骤运行项目并在浏览器中进行查看，在列表操作项中单击下载即可下载 HDFS 中文件到本地相应的目录下，如图 6-11 所示。

| 文件名 | 属性 | 大小(KB) | 操作 | 操作 |
| --- | --- | --- | --- | --- |
| 新建文本文档 (2).txt | 文件 | 2 | 删除 | 下载 |
| 第5章 项目实战：实验吧.docx | 文件 | 518 | 删除 | 下载 |
| 网盘项目需求概要文档 -.docx | 文件 | 21 | 删除 | 下载 |
| 01100160011145024543.pdf | 文件 | 57 | 删除 | 下载 |
| 13.txt | 文件 | 11 | 删除 | 下载 |
| 2 Hadoop基础环境配置.docx | 文件 | 2220 | 删除 | 下载 |
| _SUCCESS | 文件 | 0 | 删除 | 下载 |
| c16f5b41767f5acfa1c7cd8c.jpg | 文件 | 267 | 删除 | 下载 |
| part-m-00000 | 文件 | 0 | 删除 | 下载 |

图 6-11　下载操作

（17）单击右键 com.simple.bean 包名，选择“New”→“Class”新建名为“UserBean”的类文件，操作用户的属性的内容，用户的 ID、name、password 属性，重写属性的 set 与 get 方法。

```
package com.simple.bean;
public class UserBean {
    private int id;
    private String name;
    private String password;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    @Override
    public String toString() {
        return "UserBean [id=" + id + ", name=" + name + ", password="
                + password + "]";
    }
    public UserBean(int id, String name, String password) {
        super();
        this.id = id;
        this.name = name;
        this.password = password;
    }
    public UserBean() {
```

```
        super();
    }
}
```

（18）将 MySQL 的驱动包导入至 lib 文件夹下，右击 jar 包，选择“Build Path”→“Add to Build Path”在项目中添加。

（19）打开本地安装的 MySQL，在 MySQL 中新建数据库 Hadoop，通过 Navicat 工具连接 MySQL，为数据库 Hadoop 新建 student 表并添加属性，如图 6-12 所示。

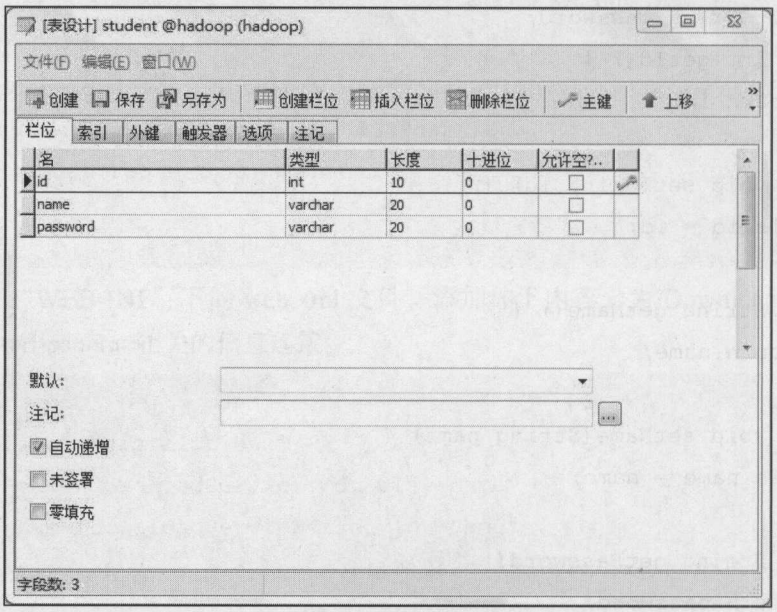



图 6-12　表属性信息

（20）右击 com.simple.model 包名，选择“New”→“Class”新建名为“ConnDB”的类文件，操作连接数据库的内容。

```
package com.simple.model;
public class ConnDB {
    private Connection ct = null;
    public Connection getConn(){
        try {
            //加载驱动
            Class.forName("com.mysql.jdbc.Driver");
            //得到连接
ct =
DriverManager.getConnection("jdbc:mysql://localhost:3306/hadoop?user=root&
password=root");
        } catch (Exception e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
        return ct;
    }
}
```

（21）单击右键 WebContent，选择“New”→“JSP File”新建名为“login.jsp”的文件，打开并编译 login.jsp 文件，操作用户的登录界面及设置监听。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript">
    function checkUser() {
        if (document.login.inputname.value == "") {
            alert("用户名不能为空.");
            return false;
        }
        if (document.login.inputpass.value == "") {
            alert("密码不能为空.");
            return false;
        }
        return true;
    }
</script>
<title>西普教育</title>
<link rel="stylesheet" href="assets/css/style.css">
<body
    style="background-color: #D3A4FF; background-position: center;
 background-repeat: repeat-y">
    <div class="login-container">
        <h1>网盘</h1>
        <div class="connect">
            <p>www.shiyanbar.com</p>
        </div>
        <form action="LoginServlet" method="post" id="loginForm" name="login"
            onsubmit="return checkUser()">
            <div>
                <input type="text" id="inputname" name="username"
 class="username"
```

```
                placeholder="用户名" autocomplete="off" />
            </div>
            <div>
                <input type="password" id="inputpass" name="password"
                    class="password" placeholder="密码" oncontextmenu="return
false"
                    onpaste="return false" />
            </div>
            <button id="submit" type="submit">登 录</button>
        </form>
        <a href="register.jsp">
            <button type="button" class="register-tis">还有没有账号? </button>
        </a>
    </div>
    <div
        style="text-align: center; margin: 50px 0; font: normal 14px/24px
 'MicroSoft YaHei';">
        <p>适用浏览器：360、FireFox、Chrome、Safari、Opera、傲游、搜狗、世界之窗. 不
支持 IE8 及以下浏览器。</p>
        <p>
            来源：<a href="http://sc.chinaz.com/" target="_blank">西普教育</a>
        </p>
    </div>
</body>
</html>
```

（22）单击右键 WebContent，选择“New”→“JSP File”新建名为“register.jsp”的文件，打开并编译 register.jsp 文件，操作用户的注册界面并对用户的操作做监听。

```
<!DOCTYPE html>
<html>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript">
function checkRegister() {
    if (document.register.inputname.value == "") {
        alert("用户名不能为空.");
        return false;
    }else{
        if (document.register.inputpass.value == "") {
            alert("密码不能为空.");
```

```
            return false;
        }else {
            if (document.register.surepass.value == "") {
                alert("确认密码不能为空.");
                return false;
            }else {
                if(document.register.inputpass.value !=
 document.register.surepass.value){
                    alert("两次密码不一致。");
                    return false;
                }
            }
        }
    }
    return true;
}
</script>
<title>注册</title>
<link rel="stylesheet" href="assets/css/login.css">
<body>
<div >
    <h1>注册</h1>
    <div class="connect">
        <p>信息注册</p>
    </div>
    <form action="RegisterServlet" method="post" id="registerForm"
 onsubmit="return checkRegister()" name="register">
        <div>
            <input type="text" name="username" id="inputname" class="username"
 placeholder="您的用户名" autocomplete="off"/>
        </div>
        <div>
            <input type="password" name="password" id="inputpass"
 class="password" placeholder="输入密码"  />
        </div>
        <div>
            <input type="password" name="surepass" id="surepass" class="surepass"
 placeholder="再次输入密码" />
        </div>
```

```
        <button id="submit" type="submit">注 册</button>
    </form>
    <a href="login.jsp">
        <button type="button" class="register-tis">已经有账号？ </button>
    </a>
</div>
</body>
</html>
```

（23）单击右键 com.simple.controller 包名，选择“New”→“Class”新建名为“LoginServlet”的类文件，并对该类进行编译，操作登录对用户的登录信息在数据库中查询，存在登录成功，否则登录失败。

```
package com.simple.controller;
public class LoginServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        this.doPost(request, response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        UserDAO user = new UserDAO();
        if(user.checkUser(username, password)){
            //用户合法，跳转到界面
            HttpSession session = request.getSession();
            session.setAttribute("username", username);
            HdfsDAO hdfs = new HdfsDAO();
            FileStatus[] documentList = hdfs.getDirectoryFromHdfs();
            request.setAttribute("documentList",documentList);
            System.out.println("得到 list 数据"+documentList);
        request.getRequestDispatcher("index.jsp").forward(request, response);
        }else{
            //用户不合法，调回登录界面，并提示错误信息
        request.getRequestDispatcher("login.jsp").forward(request,
response);
        }
    }
}
```

（24）单击右键 com.simple.model 包名，选择“New”→“Class”新建名为“UserDAO”的类文件，操作对数据验证，插入数据至数据库的内容。

```
package com.simple.model;
public class UserDAO {
    private Statement sm = null;
    private Connection ct = null;
    private ResultSet rs = null;
    public void close(){
        try {
            if(sm != null){
                sm.close();
                sm = null;
            }
            if(ct != null){
                ct.close();
                ct = null;
            }
            if(rs != null){
                rs.close();
                rs = null;
            }
        }
        catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    //检查登录用户是否合法
    public boolean checkUser(String user, String password){
        boolean b = false;
        try {
            //获得连接
            ct = new ConnDB().getConn();
            //创建 statement
            sm = ct.createStatement();
            String sql = "select * from student where name=\""+user+"\"";
            rs = sm.executeQuery(sql);
            if(rs.next()){
                //说明用户存在
```

```
                String pwd = rs.getString(3);
                if(password.equals(pwd)){
                    //说明密码正确
                    b = true;
                }else{
                    b = false;
                }
            }else{
                b = false;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            this.close();
        }
        return b;
    }
    public void insert(String name,String password) throws SQLException{
        int i = 0;
        //获得连接
        ct = new ConnDB().getConn();
        //创建 statement
        sm = ct.createStatement();
        String sql = "insert into student (name,password) values ('"+name
+"','"+password+"')";
        System.out.println(sql+"333333333");
        i = sm.executeUpdate(sql);
    }
}
```

（25）单击右键 com.simple.controller 包名，选择“New”→“Class”新建名为“RegisterServlet”的类文件，操作用户注册的信息，并将用户的信息保存至 MySQL 中。

```
package com.simple.controller;
public class RegisterServlet extends HttpServlet{
    protected void doGet(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
        this.doPost(request, response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse
```

```
response) throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        HttpSession session = request.getSession();
        session.setAttribute("username", username);
        UserDAO user = new UserDAO();
        try {
            user.insert(username, password);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        HdfsDAO hdfs = new HdfsDAO();
        FileStatus[] documentList = hdfs.getDirectoryFromHdfs();
        request.setAttribute("documentList",documentList);
        request.getRequestDispatcher("index.jsp").forward(request,
response);
    }
}
```

（26）打开“WEB-INF”下的 web.xml 文件，为 LoginServlet 和 RegisterServlet 配置<servlet-class> <url-pattern>相关的信息连接。

```
<servlet>
  <description></description>
  <display-name>LoginServlet</display-name>
  <servlet-name>LoginServlet</servlet-name>
  <servlet-class>com.simple.controller.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>LoginServlet</servlet-name>
  <url-pattern>/LoginServlet</url-pattern>
</servlet-mapping>
<servlet>
  <description></description>
  <display-name>RegisterServlet</display-name>
  <servlet-name>RegisterServlet</servlet-name>
  <servlet-class>com.simple.controller.RegisterServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>RegisterServlet</servlet-name>
```

```
    <url-pattern>/RegisterServlet</url-pattern>
  </servlet-mapping>
```

（27）单击右键 WebContent，选择“New”→“JSP File”新建名为“head.jsp”的文件，打开并编译 head.jsp 文件，对项目的界面进行美化。

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
 <link rel="stylesheet" type="text/css" href="assets/css/bootmetro.css">
   <link rel="stylesheet" type="text/css"
 href="assets/css/bootmetro-responsive.css">
   <link rel="stylesheet" type="text/css" href="assets/css/bootmetro-icons.css">
   <link rel="stylesheet" type="text/css"
 href="assets/css/bootmetro-ui-light.css">
   <link rel="stylesheet" type="text/css" href="assets/css/datepicker.css">
   <!--  these two css are to use only for documentation -->
   <link rel="stylesheet" type="text/css" href="assets/css/site.css">
   <!-- Le fav and touch icons -->
   <link rel="shortcut icon" href="assets/ico/favicon.ico">
   <link rel="apple-touch-icon-precomposed" sizes="144x144"
 href="assets/ico/apple-touch-icon-144-precomposed.png">
   <link rel="apple-touch-icon-precomposed" sizes="114x114"
 href="assets/ico/apple-touch-icon-114-precomposed.png">
   <link rel="apple-touch-icon-precomposed" sizes="72x72"
 href="assets/ico/apple-touch-icon-72-precomposed.png">
   <link rel="apple-touch-icon-precomposed"
 href="assets/ico/apple-touch-icon-57-precomposed.png">
   <!-- All JavaScript at the bottom, except for Modernizr and Respond.
     Modernizr enables HTML5 elements & feature detects; Respond is a polyfill
for min/max-width CSS3 Media Queries
     For optimal performance, use a custom Modernizr build:
 www.modernizr.com/download/ -->
   <script src="assets/js/modernizr-2.6.2.min.js"></script>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
</html>
```

（28）下载 bootmetro-master 框架，将里面的 assets 下面的文件夹导入项目的 WebContent 文件夹下面，美化用户的操作界面，如图 6-13 所示。

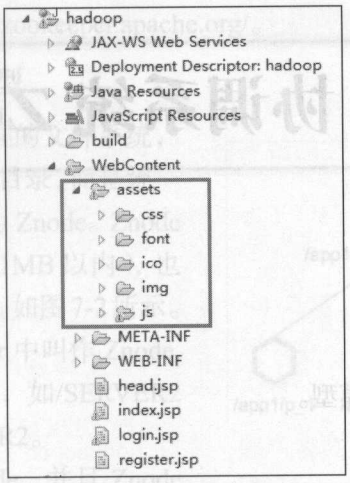



图 6-13 添加 css 等文件

（29）用相同步骤运行 Hadoop 项目，运行前需要修改 web.xml 文件中的 welcome-file 值。

```
<welcome-file-list>
  <welcome-file>login.html</welcome-file>
  <welcome-file>login.htm</welcome-file>
  <welcome-file>login.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

（30）在浏览器上查看网盘的项目效果展示，如无账号，请先进行注册，直接登录，如图 6-1 和图 6-2 所示。

## 本章小结

本章通过前面学习的知识完成一个简单的网盘案例应用，主要涉及的知识有 Linux 的基本命令操作，Hadoop 的基本命令操作，MySQL 的操作，Web 项目的增删查等。

## 习题

1. 简述云盘的原理。
2. 简述 Hadoop 基本操作。

扫一扫在线测

# 提 高 篇

# 第 7 章 协调系统 Zookeeper

## 本章要点

- 了解 Zookeeper 概述。
- 理解 Zookeeper 数据模型。
- 了解 Zookeeper 术语。
- 理解 Zookeeper 事件。
- 熟悉 Zookeeper Shell 命令。
- 熟悉 Zookeeper API 操作。

## 引　言

Zookeeper 是一个开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 HBASE 的重要组件。主要解决分布式应用一致性问题。本章通过对 Zookeeper、Zookeeper 数据模型、Zookeeper Shell 命令、Zookeeper API 操作的讲解，让学生深刻理解并学会运用 Zookeeper 系统。

## 7.1 Zookeeper 概述

### 7.1.1 Zookeeper 简介

Zookeeper 是一个能够高效开发和维护分布式的开放源码的应用协调服务。是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括维护配置信息、名字服务、分布式同步、组服务等。这些服务都被应用在分布式应用程序或其他一些形式。如图 7-1 所示。

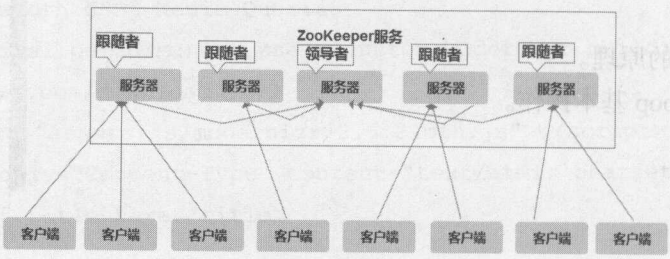



图 7-1　Zookeeper Service

源自 Google 的 Chubby 论文，发表于 2006 年 11 月，Zookeeper 是 Chubby 克隆版。协调服务是比较难的，出现 Zookeeper 之后，实现协调分布式应用就变得简单了。

Zookeeper 的官网为 http://zookeeper.apache.org/。

### 7.1.2 Zookeeper 数据模型

Zookeeper 的结构类似标准的文件系统，但这个文件系统中没有文件和目录，而是统一使用节点（node）的概念，称为 Znode。Znode 作为保存数据的容器（限制在 1MB 以内），也构成了一个层次化的命名空间。如图 7-2 所示。

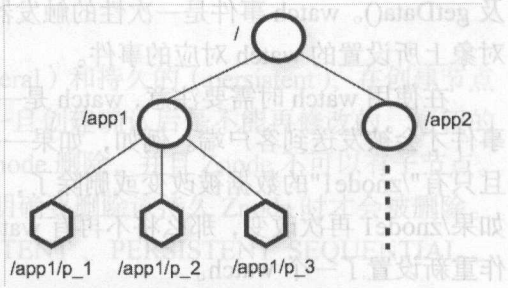



图 7-2 数据模型

（1）每个节点在 Zookeeper 中叫作 Znode，并且其有一个唯一的路径标识，如/SERVER2 节点的标识就为/APP3/SERVER2。

（2）Znode 可以有子 Znode，并且 Znode 里可以存数据，但是 EPHEMERAL 类型的节点不能有子节点。

（3）Znode 中的数据可以有多个版本，比如某一个路径下存有多个数据版本，那么查询这个路径下的数据就需要带上版本。

（4）Znode 可以是临时节点，一旦创建这个 Znode 的客户端与服务器失去联系，这个 Znode 也将自动删除，Zookeeper 的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这个连接状态称为 session，如果 Znode 是临时节点，这个 session 失效，Znode 也就删除了。

（5）Znode 的目录名可以自动编号，如 App1 已经存在，再创建的话，将会自动命名为 App2。

（6）Znode 可以被监控，包括这个目录节点中存储的数据的修改，子节点目录的变化等，一旦变化可以通知设置监控的客户端，这个功能是 Zookeeper 对于应用最重要的特性，通过这个特性可以实现的功能包括配置的集中管理、集群管理、分布式锁等。

### 7.1.3 Zookeeper 特征

（1）Zookeeper 的数据模型。

Zookeeper 拥有一个层次的命名空间。这就是一个文件系统，只不过文件系统中的文件还可以具有目录的功能。另外，指向节点的路径必须用规范的绝对路径来表示，并且以斜线“/”来分割。

（2）Zookeeper 会话及状态。

Zookeeper 客户端通过句柄为 Zookeeper 服务建立一个会话。这个会话一旦创建，句柄将以 CONNECTING 状态开始启动。客户端尝试连接到其中一个 Zookeeper 服务器，如果连接成功，它的状态将变为 CONNECTED。在一般情况下只有上述这两种状态。如果一个可恢复的错误发生，比如会话终结或认证失败，或者应用程序明确地关闭了句柄，句柄将会转入关闭状态。

为了创建一个客户端会话，应用程序必须提供一个由主机（IP 或主机名）和端口所组

成的连接字符串，这个字符串标识了要连接的目标主机及主机端口。Zookeeper 客户端将选择服务器列表中的任意一个服务器并尝试连接。如果连接失败，那么客户端将自动尝试连接服务列表中的其他服务器，直到连接成功。

（3）Zookeeper watches。

Zookeeper 可以为所有的读操作设置 watch，这些读操作包括 exists()、getChildren()以及 getData()。watch 事件是一次性的触发器，当 watch 的对象状态发生改变时，将会触发此对象上所设置的 watch 对应的事件。

在使用 watch 时需要注意，watch 是一次性触发器，并且只有在数据发送改变时，watch 事件才会被发送到客户端。例如，如果一个客户端进行了 getData("/znode1", true)操作，并且只有"/znode1"的数据被改变或删除了，那么客户端将获得一个关于“znode1”的时间。如果/znode1 再次改变，那么将不再有 watch 事件发生给客户端，除非客户端为另一个读操作重新设置了一个 watch。

watch 事件将被异步地发送到客户端，并且 Zookeeper 为 watch 机制提供了有序的一致性保证。理论上，客户端接收 watch 的事件的时间要快于其看到 watch 对象状态变化的时间。

（4）Zookeeper ACL。

Zookeeper 使用 ACL 来对 Znode 进行访问控制。ACL 的实现和 UNIX 文件访问许可非常相似：它使用许可位来对一个节点的不同操作进行运行或者禁止的权限控制。但是，和标准 UNIX 不同的是，Zookeeper 节点有 user（文件的拥有者）、group 和 world 三种标准模式，并且没有节点所有者的概念。

（5）Zookeeper 的一致性保证。

Zookeeper 是一种高性能、可扩展的服务。Zookeeper 的读写速度非常快，并且读的速度要比写更快。另外，在进行读操作的时候，Zookeeper 依然能够为旧数据提供服务。这些都是由 Zookeeper 所提供的一致性保证的。

### 7.1.4 Zookeeper 工作原理

（1）Zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫作 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

（2）一旦 leader 已经和多数的 follower 进行了状态同步后，它就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 Zookeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。Zookeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

（3）广播模式需要保证 proposal 被按顺序处理，因此 zk 采用了递增的事务 id 号（zxid）来保证。所有的提议（proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它

都会有一个新的 epoch。低 32 位是个递增计数。

（4）当 leader 崩溃或者 leader 失去大多数的 follower，这时候 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态。

## 7.2 Zookeeper 术语

### 7.2.1 节点

Znode 节点主要有两种类型，短暂的（ephemeral）和持久的（persistent），在创建节点时可以根据需要进行创建，但是 Znode 的类型一旦创建确认后是不能再修改的。短暂的 Znode 的客户端结束时，Zookeeper 会将该短暂 Znode 删除，并且 Znode 不可以有子节点。持久 Znode 不依赖于客户端会话，只有当客户端明确要删除该持久 Znode 时才会被删除。Znode 有四种形式的目录节点，即 PERSISTENT、PERSISTENT_SEQUENTIAL、EPHEMERAL、EPHEMERAL_SEQUENTIAL。

### 7.2.2 角色

在 Zookeeper 中，是有角色的概念的，主要分为：①领导者（leader）角色，负责进行投票的发起和决议，更新系统状态；②学习者（learner）角色，包括跟随者（follower）和观察者（observer），follower 用于接受客户端请求并向客户端返回结果，在选领导者过程中参与投票。observer 可以接受客户端连接，将写请求转发给 leader，但 observer 不参加投票过程，只同步 leader 的状态，observer 的目的是为了扩展系统，提高读取速度。

### 7.2.3 顺序号

在创建 Znode 时需要设置顺序标识，并且 Znode 名称后会附加一个值。顺序号是一个单调递增的计数器，由父节点维护。在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序。

### 7.2.4 观察

Watcher 在 Zookeeper 中是一个核心功能，Watcher 可以监控目录节点的数据变化以及子目录的变化，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 Watcher，从而每个客户端都很快知道它所关注的目录节点的状态发生变化，而做出相应的反应。

### 7.2.5 Leader 选举

在 Zookeeper 中，每个 Server 启动以后都询问其他的 Server 它要投票给谁。对于其他 server 的询问，server 每次根据自己的状态都回复自己推荐的 leader 的 id 和上一次处理事务的 zxid，收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。计算这过程中获得票数最多的 Server 为获胜者，如果获胜者的票数超过半数，则该 Server 被选为 Leader。否则，继续这个过程，直到 Leader 被选举出来。之后 Leader 就会开始等待 Server 连接，Follower 连接 Leader，将最大的 zxid 发送给 Leader，Leader 根据 follower 的 zxid 确定同步点，完成同步后通知 Follower 已经

成为 uptodate 状态，Follower 收到 uptodate 消息后，又可以重新接受 Client 的请求进行服务了。

## 7.3 事件

Zookeeper 的节点是可以被监控，目录中存储数据的修改、子节点目录的变化，都可以触发事件并通知监听的客户端，这是 Zookeeper 重要的特性，通过此特性可以实现的功能有配置的集中管理、集群管理、分布式锁等。监听机制官方说明为：一个监听事件是一个一次性的监听器，当被设置了监听的数据发生变化时，服务器就会将这个改变发送给负责设置 Watch 的客户端。

Zookeeper 的特点是:

- 一次性触发。
- 事件触发后发给客户端。
- 数据被设置 Watch。

## 7.4 Zookeeper Shell 操作

Zookeeper 命令行工具类似于 Linux 的 Shell 环境，能够实现简单地对 Zookeeper 进行访问、数据创建、数据修改等操作，在操作之前需要使用 zkCli.sh -server 127.0.0.1:2181 连接到 Zookeeper 服务，连接成功后，系统会输出 Zookeeper 的相关环境以及配置信息。最后就可以通过输入一些命令，操作 Zookeeper。

### 7.4.1 Zookeeper 服务命令

Zookeeper 服务安装之后，一般会在这个服务的基础之上安装其他的大数据平台，其他的框架一般会提供很多接口对 Zookeeper 中的内容进行一定的操作，但是功能相对单一，所以有些时候，有必要我们自己登录 Zookeeper 服务器，对里面的文件结构有一定的了解，这样的话使用起来也比较方便。首先是最基本的 zkServer.sh 脚本的使用，主要的操作命令如下：

- 启动 ZK 服务　　sh bin/zkServer.sh start
- 查看 ZK 服务状态　　sh bin/zkServer.sh status
- 停止 ZK 服务　　sh bin/zkServer.sh stop
- 重启 ZK 服务　　sh bin/zkServer.sh restart

【案例 7-1】Zookeeper 伪集群模式配置

为完全发挥并提高 Zookeeper 的性能，在安装部署过程中需要采用伪集群或集群部署。实验中主要以 Linux 平台为主介绍 Zookeeper 的集群部署。

（1）Zookeeper 软件存放在 soft 文件下,在 soft 目录下，执行命令 ls，查看该目录下的文件列表，如图 7-3 所示。

（2）利用 tar 执行命令 tar -zxvf /soft/zookeeper-3.4.5.tar.gz -C /simple,解压 Zookeeper 到 simple 目录下，如图 7-4 所示。

（3）在 simple 下,执行命令 ls，查看解压的文件，如图 7-5 所示。

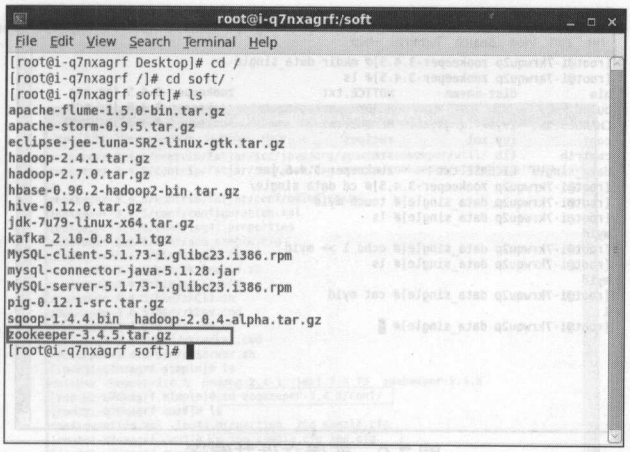



图 7-3　软件位置

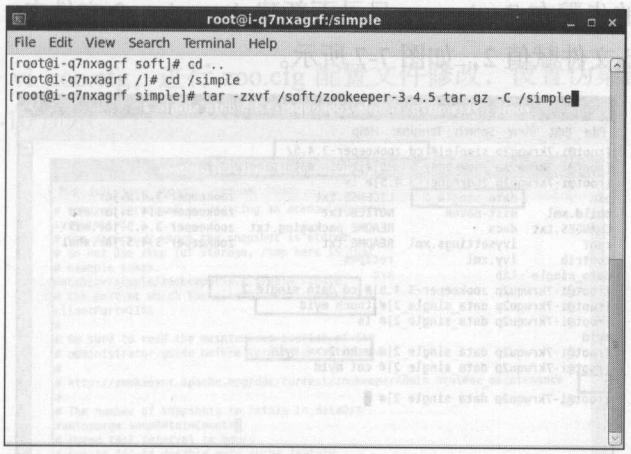



图 7-4　解压 Zookeeper

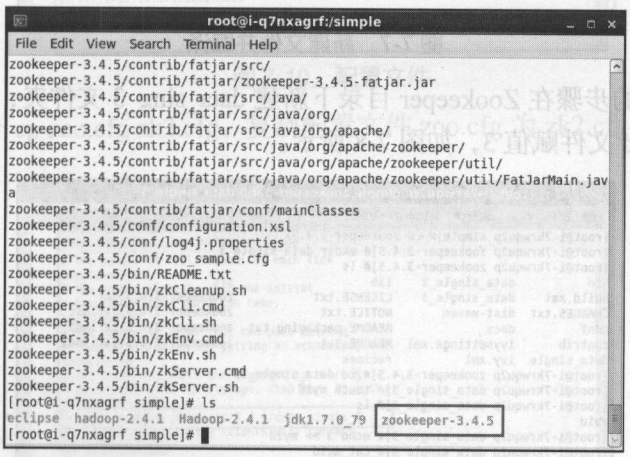



图 7-5　查看 Zookeeper

（4）通过命令 cd zookeeper-3.4.5 进入到 Zookeeper 目录下，执行命令 mkdir data_single，新建 data_single 文件夹，在文件夹下执行命令 touch myid，新建 myid 文件,对文件进行编译 echo 1 >> myid，如图 7-6 所示。

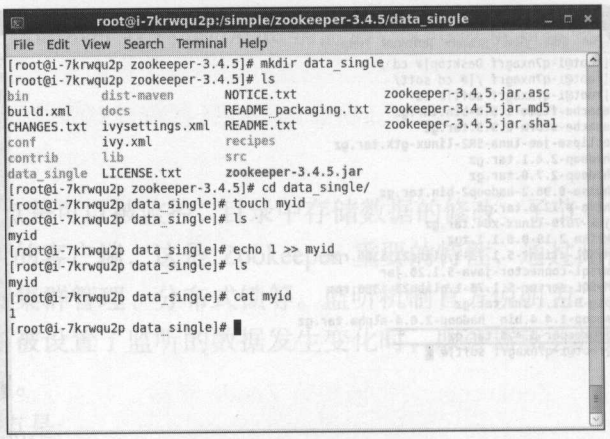



图 7-6　新建文件并编译

（5）通过相同的步骤在 Zookeeper 目录下新建 data_sing_2 文件夹，并在文件夹下新建 myid 文件，为 myid 文件赋值 2，如图 7-7 所示。

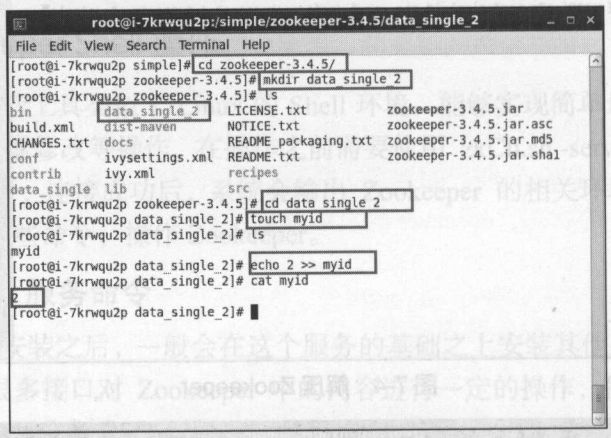



图 7-7　新建文件并编译

（6）通过相同的步骤在 Zookeeper 目录下新建 data_sing_3 文件夹，并在文件夹下新建 myid 文件，为 myid 文件赋值 3，如图 7-8 所示。

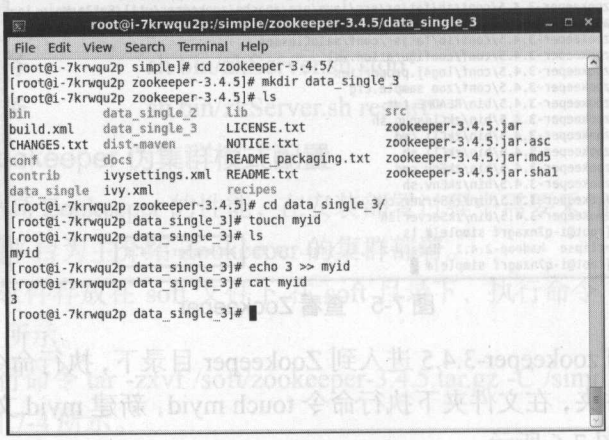



图 7-8　新建文件并编译

（7）进入到 zookeeper-3.4.5/conf 目录下，复制 zoo_sample.cfg 文件并改名 zoo.cfg，如图 7-9 所示。

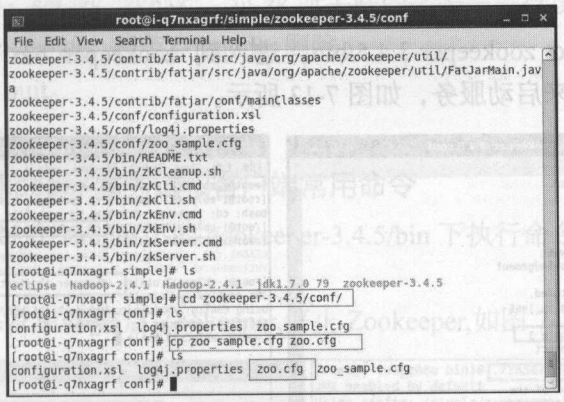



图 7-9　复制文件并改名

（8）执行命令 vi zoo.cfg，进行 zoo.cfg 配置文件修改，设置伪集群配置参数，添加伪集群配置，如图 7-10 所示。

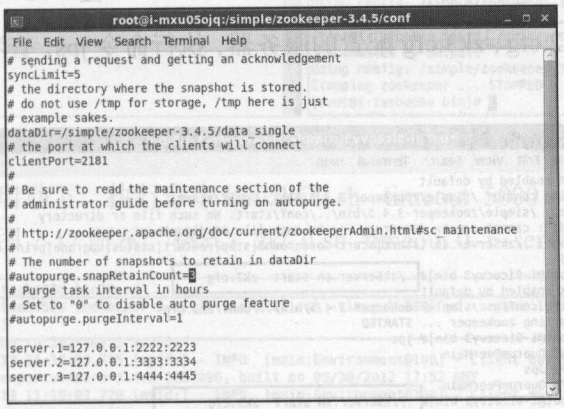



图 7-10　配置文件

（9）执行命令 cp zoo.cfg zk2.cfg，复制配置文件 zoo.cfg 为 zk2.cfg 并修改端口为 2182，如图 7-11 所示。

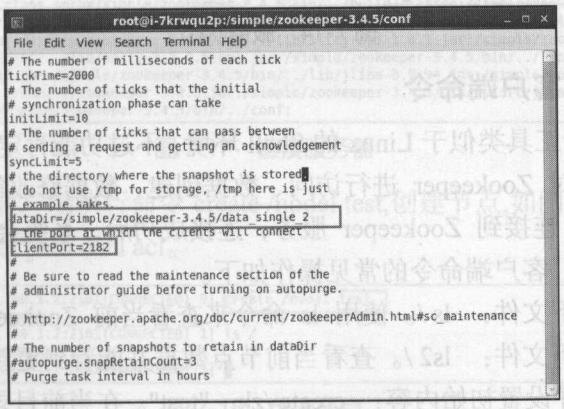



图 7-11　端口修改

（10）执行命令 cp zoo.cfg zk3.cfg，复制配置文件 zoo.cfg 为 zk3.cfg 并修改端口为 2183，如图 7-12 所示。

（11）启动集群。

通过执行命令 cd zookeeper-3.4.5/bin，进入到 zookeeper bin 文件下，通过执行命令./zkServer.sh start，来启动服务，如图 7-13 所示。

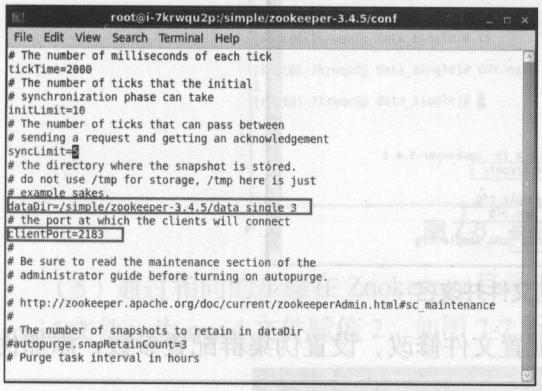



图 7-12　端口修改

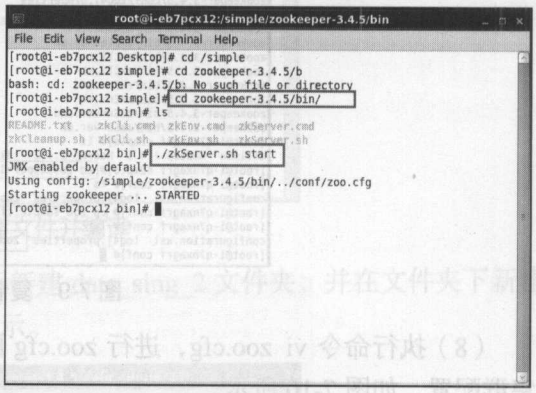



图 7-13　启动集群

（12）同理启动 zk2.cfg、zk3.cfg 配置的服务器，执行命令 jps，查看进程列表，如图 7-14 所示。

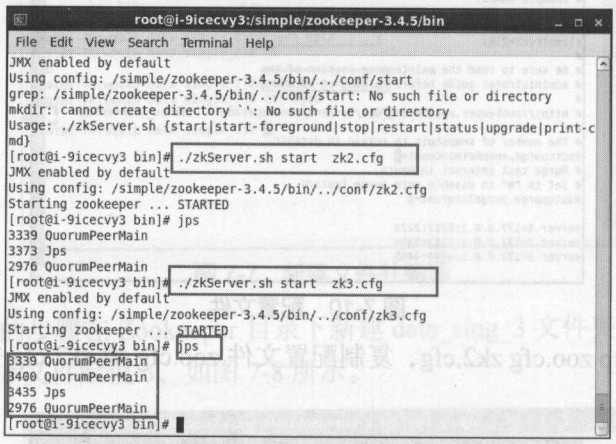



图 7-14　服务状态

## 7.4.2　Zookeeper 客户端命令

Zookeeper 命令行工具类似于 Linux 的 Shell 环境，不过功能肯定不及 Shell 啦，但是使用它我们可以简单地对 Zookeeper 进行访问、数据创建、数据修改等操作. 使用 zkCli.sh -server 127.0.0.1:2181 连接到 Zookeeper 服务，连接成功后，系统会输出 Zookeeper 的相关环境以及配置信息。客户端命令的常见操作如下。

- 显示根目录下文件： ls /。使用 ls 命令来查看当前 Zookeeper 中所包含的内容。
- 显示根目录下文件： ls2 /。查看当前节点数据并能看到更新次数等数据。
- 创建文件，并设置初始内容： create /zk "test"。在当前目录创建一个新的 znode

节点“ zk ”以及与它关联的字符串。

- 获取文件内容：get /zk。获取 zk 所包含的信息。
- 修改文件内容：set /zk "zkbak"。对 zk 所关联的字符串进行设置。
- 删除文件：delete /zk。将节点 znode 删除。
- 退出客户端：quit。
- 帮助命令：help。

**【案例 7-2】Zookeeper 服务端和客户端常用命令**

（1）启动 Zookeeper,在 simple 下的 zookeeper-3.4.5/bin 下执行命令./zkServer.sh start,如图 7-15 所示。

（2）查看服务状态及集群,重启 Zookeeper,停止 Zookeeper,如图 7-16 所示。

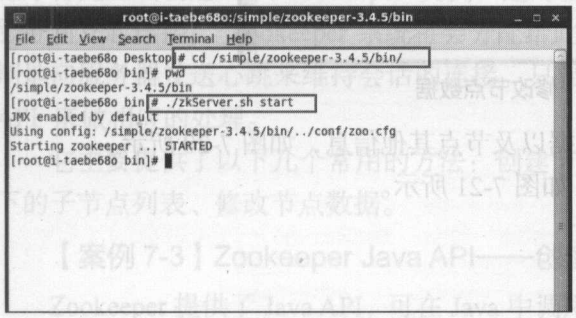



图 7-15　启动 Zookeeper

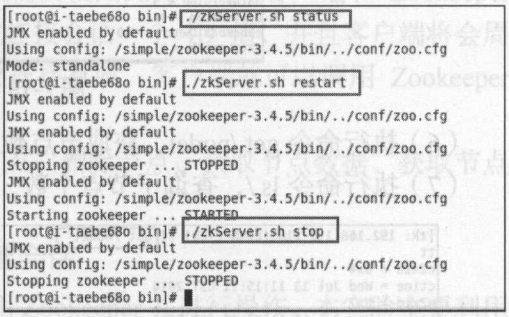



图 7-16　重启 Zookeeper

（3）在 Zookeeper 启动下，执行命令 bin/zkCli.sh -timeout 5000 -r -server 192.168.1.2:2181，连接服务器，如图 7-17 所示。格式：zkCli.sh -timeout 0 -r -server ip:port。

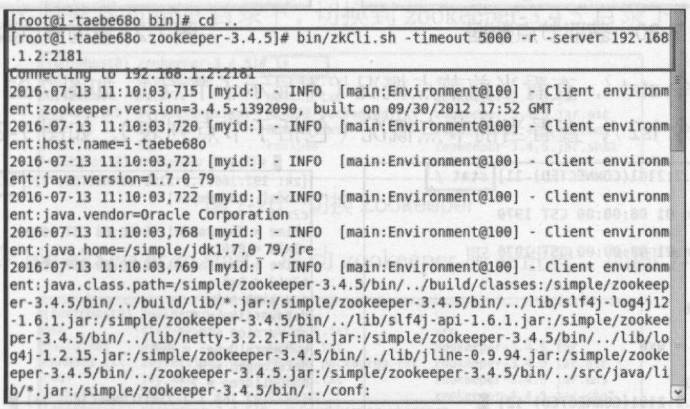



图 7-17　链接服务器

（4）在 Zookeeper 客户端执行命令 create /nodel test,创建节点,如图 7-18 所示。格式：create [-s] [-e] path data acl。

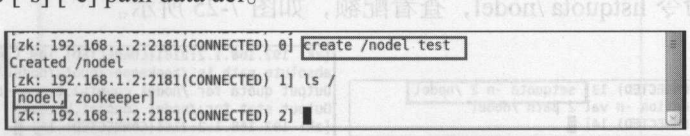



图 7-18　创建节点

（5）如果想修改已存在节点的数据，可以通过执行命令 set /nodel tt，修改节点数据，如图 7-19 所示。

```
File  Edit  View  Search  Terminal  Help
[zk: 192.168.1.2:2181(CONNECTED) 6] get /nodel
test
cZxid = 0x4
ctime = Wed Jul 13 11:15:11 CST 2016
mZxid = 0x6
mtime = Wed Jul 13 11:19:48 CST 2016
pZxid = 0x4
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 4
numChildren = 0
[zk: 192.168.1.2:2181(CONNECTED) 7] set /nodel tt
cZxid = 0x4
ctime = Wed Jul 13 11:15:11 CST 2016
mZxid = 0x7
mtime = Wed Jul 13 11:20:01 CST 2016
pZxid = 0x4
cversion = 0
dataVersion = 3
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 2
```

图 7-19　修改节点数据

（6）执行命令 get /nodel，获取节点数据以及节点其他信息，如图 7-20 所示。

（7）执行命令 ls /，查询子节点个数，如图 7-21 所示。

```
[zk: 192.168.1.2:2181(CONNECTED) 8] get /nodel
tt
cZxid = 0x4
ctime = Wed Jul 13 11:15:11 CST 2016
mZxid = 0x7
mtime = Wed Jul 13 11:20:01 CST 2016
pZxid = 0x4
cversion = 0
dataVersion = 3
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 2
numChildren = 0
[zk: 192.168.1.2:2181(CONNECTED) 9]
```

图 7-20　获取节点数据

```
[zk: 192.168.1.2:2181(CONNECTED) 10] ls /
[nodel, zookeeper]
[zk: 192.168.1.2:2181(CONNECTED) 11]
```

图 7-21 查询节点数据

（8）执行命令 stat /，查看当前节点概况（不显示子节点），如图 7-22 所示。

（9）执行命令 ls2 /，查看当前节点概况（包括子节点列表），如图 7-23 所示。

```
[zk: 192.168.1.2:2181(CONNECTED) 11] stat /
cZxid = 0x0
ctime = Thu Jan 01 08:00:00 CST 1970
mZxid = 0x0
mtime = Thu Jan 01 08:00:00 CST 1970
pZxid = 0x4
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 2
[zk: 192.168.1.2:2181(CONNECTED) 12]
```

图 7-22　查看节点概况

```
[zk: 192.168.1.2:2181(CONNECTED) 12] ls2 /
[nodel, zookeeper]
cZxid = 0x0
ctime = Thu Jan 01 08:00:00 CST 1970
mZxid = 0x0
mtime = Thu Jan 01 08:00:00 CST 1970
pZxid = 0x4
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 2
[zk: 192.168.1.2:2181(CONNECTED) 13]
```

图 7-23　查看节点列表

（10）执行命令 setquota -n 2 /nodel，设置配额，如图 7-24 所示。

（11）执行命令 listquota /nodel，查看配额，如图 7-25 所示。

```
[zk: 192.168.1.2:2181(CONNECTED) 13] setquota -n 2 /nodel
Comment: the parts are option -n val 2 path /nodel
[zk: 192.168.1.2:2181(CONNECTED) 14]
```

图 7-24　设置配额

```
[zk: 192.168.1.2:2181(CONNECTED) 14] listquota /nodel
absolute path is /zookeeper/quota/nodel/zookeeper_limits
Output quota for /nodel count=2,bytes=-1
Output stat for /nodel count=1,bytes=2
[zk: 192.168.1.2:2181(CONNECTED) 15]
```

图 7-25　查看配额

## 7.5　Zookeeper API 操作

Zookeeper API 共包含 5 个包，分别为：

- org.apache.zookeeper
- org.apache.zookeeper.data
- org.apache.zookeeper.server
- org.apache.zookeeper.server.quorum
- org.apache.zookeeper.server.upgrade

其中 org.apache.zookeeper 包含 Zookeeper 类，它是我们编程时最常用的类文件。

Zookeeper 类是 Zookeeper 客户端库的主要类文件，如果要使用 Zookeeper 服务，应用程序首先必须创建一个 Zookeeper 实例，这时就需要使用此类。一旦客户端和 Zookeeper 服务建立起连接， Zookeeper 系统将会分配给此连接会话一个 ID 值，并且客户端将会周期地向服务器发送心跳来维持会话的连接。只要连接有效，客户端就可以调用 Zookeeper API 来做相应的处理。

它主要提供了以下几个常用的方法：创建节点、删除节点、获取节点数据、获取节点下的子节点列表、修改节点数据。

### 【案例 7-3】Zookeeper Java API——创建会话

Zookeeper 提供了 Java API，可在 Java 中调用 Zookeeper 并进行操作。本实验主要利用 Zookeeper Java API 创建的 Zookeeper 对象创建连接会话。但由于 Zookeeper 对象创建会话时是异步操作，所以需要程序等待延迟关闭并在实现 watcher 接口的方法中收集连接会话后返回信息。

（1）进入 Linux 服务器/simple 目录下，切换到 zookeeper-3.4.5 目录下，如图 7-26 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt              src
build.xml    docs             NOTICE.txt               zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt     zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt               zookeeper-3.4.5.jar.md5
contrib      lib              recipes                  zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]#
```

图 7-26　切换 Zookeeper

（2）执行命令 bin/zkServer.sh start，启动 zookeeper 服务进程，如图 7-27 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt              src
build.xml    docs             NOTICE.txt               zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt     zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt               zookeeper-3.4.5.jar.md5
contrib      lib              recipes                  zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]#
```

图 7-27　启动 Zookeeper

（3）执行命令 bin/zkServer.sh status，查看服务是否启动,如图 7-28 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh status
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: standalone
[root@host01 zookeeper-3.4.5]#
```

图 7-28　查看服务状态

（4）执行命令 bin/zkCli.sh -timeout 5000 -server 192.168.1.132:2181，连接测试 Zookeeper 服务。如图 7-29 所示。

```
Mode: standalone
[root@host01 zookeeper-3.4.5]# ./bin/zkCli.sh -timeout 5000 -server 192.168.0.201:2181
Connecting to 192.168.0.201:2181
2016-07-13 13:59:41,733 [myid:] - INFO  [main:Environment@100] - Client environment:zook
eeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:host
.name=host01
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:java
.version=1.7.0_79
2016-07-13 13:59:41,738 [myid:] - INFO  [main:Environment@100] - Client environment:java
.vendor=Oracle Corporation
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.home=/simple/jdk1.7.0_79/jre
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.class.path=/simple/zookeeper-3.4.5/bin/../build/classes:/simple/zookeeper-3.4.5/bin/../
build/lib/*.jar:/simple/zookeeper-3.4.5/bin/../lib/slf4j-log4j12-1.6.1.jar:/simple/zooke
eper-3.4.5/bin/../lib/slf4j-api-1.6.1.jar:/simple/zookeeper-3.4.5/bin/../lib/netty-3.2.2
.Final.jar:/simple/zookeeper-3.4.5/bin/../lib/log4j-1.2.15.jar:/simple/zookeeper-3.4.5/b
in/../lib/jline-0.9.94.jar:/simple/zookeeper-3.4.5/bin/../zookeeper-3.4.5.jar:/simple/zo
okeeper-3.4.5/bin/../src/java/lib/*.jar:/simple/zookeeper-3.4.5/bin/../conf:
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.library.path=/usr/java/packages/lib/i386:/lib:/usr/lib
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.io.tmpdir=/tmp
```

图 7-29　连接 Zookeeper

（5）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”，新建一个项目“zkTest_1”，如图 7-30 所示。

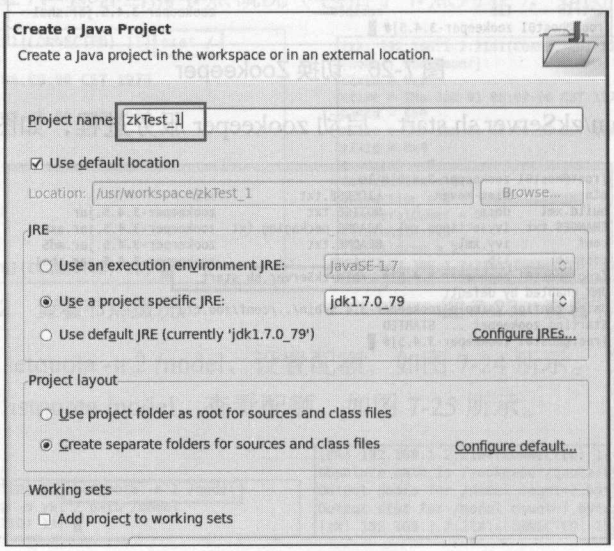



图 7-30　新建项目

（6）在项目上通过单击右键，选择“Build Path”→“Configure Build Path”→“Libraries”→“add External JARs”，选择目录文件到/simple/zookeeper-3.4.5/lib 下的 jar 包，添加相关 jar 即可（共 6 个 jar），如图 7-31 所示。

注：分别添加 zookeeper.jar 和 lib 目录下的所有 jar 包。

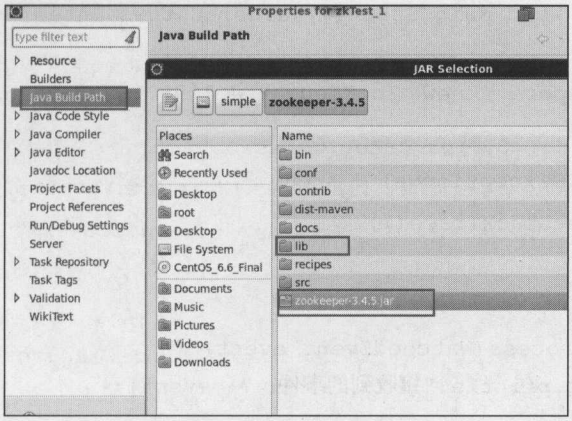



图 7-31　添加 jar 包

（7）创建相关包“com.zktest”及类“CreateSession”，如图 7-32 所示。

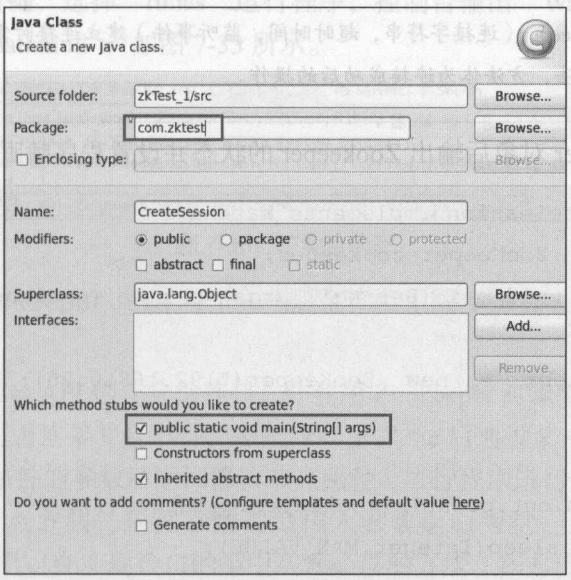



图 7-32　新建包名

（8）在类中实现 Watcher 接口，并实现其方法 process(WatchedEvent event)，如下所示。

```
public class CreateSession implements Watcher{
    public void process(WatchedEvent event) {
        System.out.println("接收到的事件: "+ event);
    }
}
```

（9）声明 Zookeeper 对象静态属性，创建 main 方法，并在 main 方法中实例化 Zookeeper 对象，如下所示。

```
public class CreateSession implements Watcher{
    private static ZooKeeper zookeeper;
    public static void main(String[] args) throws InterruptedException {
        try {
            zookeeper = new ZooKeeper( # "192.168.0.201:2181", 5000, new
CreateSession());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void process(WatchedEvent event) {
        System.out.println("接收到的事件: "+ event);
    }
}
```

**说明**

Zookeeper（连接字符串，超时时间，监听事件）建立连接时需要实现 watcher 接口，并实现其方法，方法体为连接成功后的操作。

（10）在 Zookeeper 对象后输出 Zookeeper 的状态并设置程序睡眠状态，如下所示。

```
public class CreateSession implements Watcher{
    private static ZooKeeper zookeeper;
    public static void main(String[] args) throws InterruptedException {
        try {
            zookeeper = new ZooKeeper("192.168.0.201:2181", 5000, new
CreateSession());

            System.out.println(zookeeper.getState());
            Thread.sleep(Integer.MAX_VALUE);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void process(WatchedEvent event) {
    }
}
```

注：由于连接 Zookeeper 是异步连接，需要等待一段时间后才能有返回。

（11）在实现 watcher 的 process 方法中输出接收到的事件，如下所示。

```
public class CreateSession implements Watcher{
    private static ZooKeeper zookeeper;
    public static void main(String[] args) throws InterruptedException {
        try {
            zookeeper = new ZooKeeper("192.168.0.201:2181", 5000, new
CreateSession());
            System.out.println(zookeeper.getState());
            Thread.sleep(Integer.MAX_VALUE);
        } catch (IOException e) {
            e.printStackTrace();
        }}
    public void process(WatchedEvent event) {
        System.out.println("接收到的事件: "+ event);
    }
}
```

（12）单击鼠标右键，选择“runas”运行程序，控制台输出“WatchedEvent state:SyncConnected type:None path:null”，如图 7-33 所示。

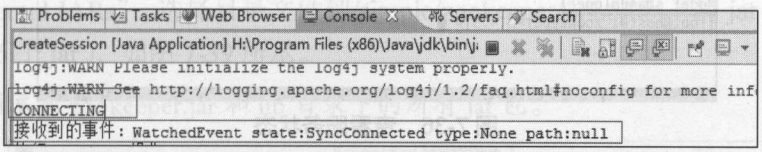



图 7-33　控制台信息

## 【案例 7-4】Zookeeper Java API——删除节点（同步）

需求描述：

通过 Zookeeper Java API 创建会话连接，并利用 Zookeeper 对象创建节点。同创建节点类似，判断当前连接是否连接并且事件类型是否等于 None 后通过 Zookeeper 对象调用删除节点方法，传入满足条件的参数后返回修改节点路径，执行程序进行删除。利用 Zookeeper 对象调用异步删除节点方法，根据参数类型传入参数返回为空，其中需要传入实现了 stringCallBack 接口的类。在实现接口的类中实现必需的方法，在方法体中收集异步删除节点事件返回的值。

（1）进入 Linux 命令终端的/simple 目录下，切换到 zookeeper-3.4.5 目录下,并执行命令 ls ,查看 zookeeper-3.4.5 下的文件列表，如图 7-34 所示。

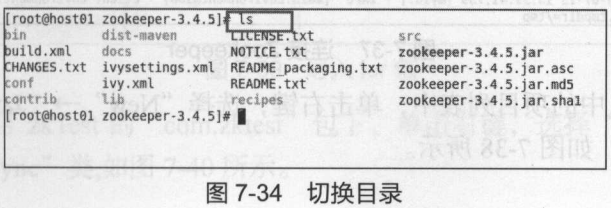



图 7-34　切换目录

（2）执行命令 bin/zkServer.sh start，启动 Zookeeper 服务进程，如图 7-35 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]#
```

图 7-35　启动服务

（3）执行命令 bin/zkServer.sh status，查看服务是否启动，如图 7-36 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh status
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: standalone
[root@host01 zookeeper-3.4.5]#
```

图 7-36　查看服务状态

（4）执行命令 bin/zkCli.sh -timeout 5000 -server 192.168.1.132:2181，连接测试 Zookeeper 服务。如图 7-37 所示。

```
Mode: standalone
[root@host01 zookeeper-3.4.5]# ./bin/zkCli.sh -timeout 5000 -server 192.168.0.201:2181
Connecting to 192.168.0.201:2181
2016-07-13 13:59:41,733 [myid:] - INFO  [main:Environment@100] - Client environment:zook
eeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:host
.name=host01
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:java
.version=1.7.0_79
2016-07-13 13:59:41,738 [myid:] - INFO  [main:Environment@100] - Client environment:java
.vendor=Oracle Corporation
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.home=/simple/jdk1.7.0_79/jre
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.class.path=/simple/zookeeper-3.4.5/bin/../build/classes:/simple/zookeeper-3.4.5/bin/../
build/lib/*.jar:/simple/zookeeper-3.4.5/bin/../lib/slf4j-log4j12-1.6.1.jar:/simple/zooke
eper-3.4.5/bin/../lib/slf4j-api-1.6.1.jar:/simple/zookeeper-3.4.5/bin/../lib/netty-3.2.2
.Final.jar:/simple/zookeeper-3.4.5/bin/../lib/log4j-1.2.15.jar:/simple/zookeeper-3.4.5/b
in/../lib/jline-0.9.94.jar:/simple/zookeeper-3.4.5/bin/../zookeeper-3.4.5.jar:/simple/zo
okeeper-3.4.5/bin/../src/java/lib/*.jar:/simple/zookeeper-3.4.5/bin/../conf:
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.library.path=/usr/java/packages/lib/i386:/lib:/usr/lib
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.io.tmpdir=/tmp
```

图 7-37　连接 Zookeeper

（5）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”，新建一个项目“zkTest_1”，如图 7-38 所示。

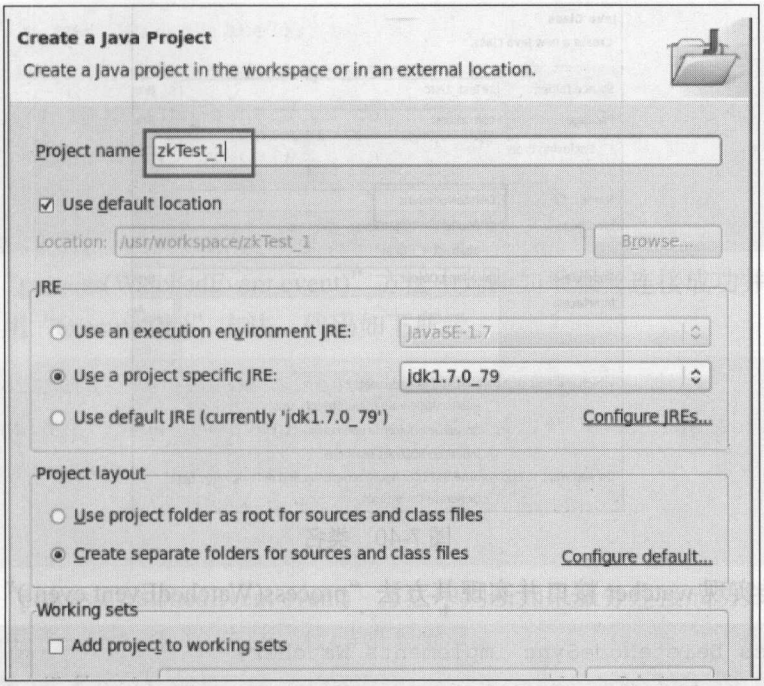



图 7-38　新建项目

（6）在项目上通过单击右键，选择“Build Path”→“Configure Build Path”→ “Libraries”→ “add External JARs”，选择目录文件到/simple/zookeeper-3.4.5/lib 下的 jar 包，添加相关 jar 即可（共 6 个 jar），如图 7-39 所示。

注：分别添加 zookeeper.jar 和 lib 目录下的所有 jar 包。

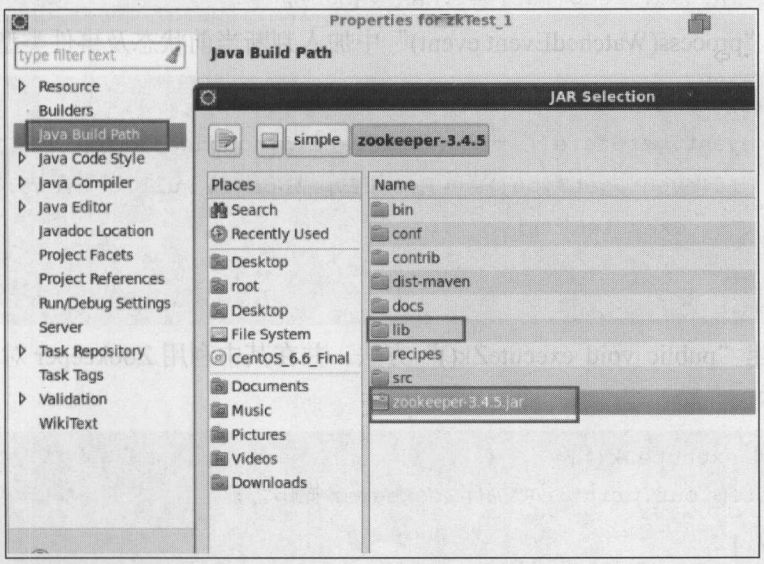



图 7-39　导入 jar 包

（7）在当前项目 zkTest 的“com.zktest”包下，单击右键，选择“New”→“Class”，创建“DeleteNodeSync”类,如图 7-40 所示。

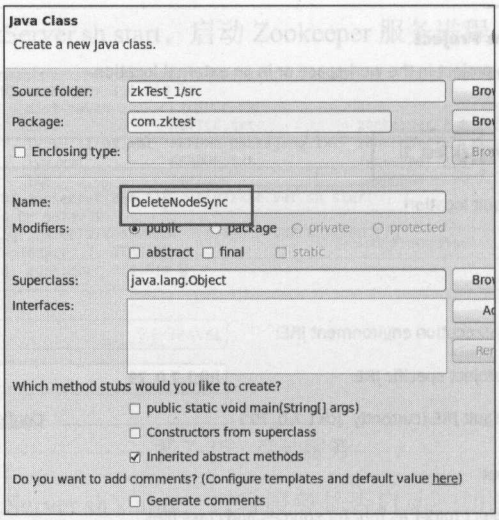



图 7-40　类名

（8）将类实现 watcher 接口并实现其方法“process(WatchedEvent event)”，如下所示。

```
public class DeleteNodeSync implements Watcher{
    public void process(WatchedEvent event) {
    }
}
```

（9）利用 Zookeeper 对象创建会话并设置连接 Zookeeper 后休眠，如下所示。

```
zookeeper = new ZooKeeper("192.168.1.132:2181", 5000, new CreateNodeASync());
            Thread.sleep(Integer.MAX_VALUE);
```

（10）在“process(WatchedEvent event)”中加入判断当前状态及事件类型，如下所示。

```
System.out.println("接收到的事件: "+ event);
        if(event.getState() == KeeperState.SyncConnected){
            if(event.getType() == EventType.None&& null == event.getPath()){
                executeZk();
            }
        }
```

（11）创建“public void executeZk()”方法，并在其中利用 Zookeeper 对象删除节点，代码如下所示。

```
public void executeZk(){
        System.out.println("执行 Zookeeper 操作");
        try {
            zookeeper.delete("/node3", -1);
        } catch (KeeperException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

（12）在“process(WatchedEvent event)”方法中判断如果服务连接成功并且事件类型返回 None 则调用“executeZk()”方法，代码如下所示。

```
if(event.getState() == KeeperState.SyncConnected){
    if(event.getType() == EventType.None&& null == event.getPath()){
        executeZk();
    }
}
```

（13）在“process(WatchedEvent event)”方法中判断如果服务连接成功并且事件类型返回 None 则调用“executeZk()”方法，代码如下所示。

```
package com.zktest;
public class DeleteNodeSync implements Watcher{
    public static  ZooKeeper zookeeper =null;
    public static void main(String[] args) throws Exception {
        zookeeper = new ZooKeeper("192.168.0.201:2181", 5000, new
DeleteNodeSync());
        Thread.sleep(Integer.MAX_VALUE);
    }
    @Override
    public void process(WatchedEvent event) {
        System.out.println("接收到的事件: "+ event);
        if(event.getState() == KeeperState.SyncConnected){
            if(event.getType() == EventType.None&& null == event.getPath()){
                executeZk();
            }
        }
    }
    public void executeZk(){
        System.out.println("执行 Zookeeper 操作");
        try {
            zookeeper.delete("/node3", -1);
        } catch (KeeperException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

（14）单击右键类名，选择“Run as”→“Java Application”运行程序，返回同步删除节点路径，结果如图 7-41 所示。

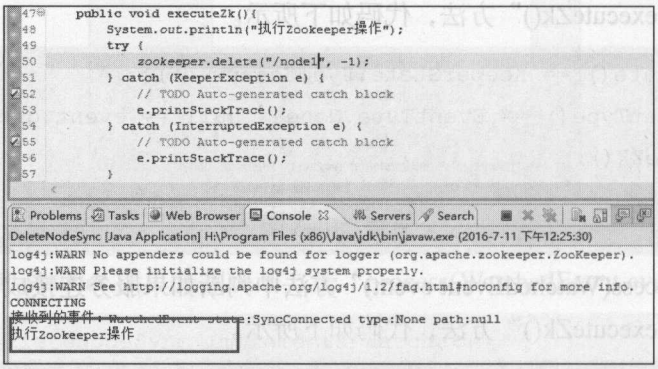



图 7-41　控制台信息

（15）进入 Linux 命令终端，执行 Zookeeper 脚本文件 zkCli.sh 连接服务，并执行命令 ls /,查看根节点下是否存在 node3,如图 7-42 所示。

```
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
[zk: 192.168.0.201:2181(CONNECTED) 8] ls /
[consumers, config, node3, brokers, admin, zookeeper, controller_epoch]
[zk: 192.168.0.201:2181(CONNECTED) 9] ls /
[consumers, config, node3, brokers, admin, zookeeper, controller_epoch]
[zk: 192.168.0.201:2181(CONNECTED) 10] get /node3
123456
cZxid = 0xa2
ctime = Wed Jul 13 15:04:45 CST 2016
mZxid = 0xb4
mtime = Wed Jul 13 16:56:01 CST 2016
pZxid = 0xa2
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
[zk: 192.168.0.201:2181(CONNECTED) 11] ls /
[consumers, config, brokers, admin, zookeeper, controller_epoch]
[zk: 192.168.0.201:2181(CONNECTED) 12]
```

图 7-42　查看根节点

## 【案例 7-5】Zookeeper Java API——修改节点（同步）

（1）进入 Linux 服务器的/simple 目录下，切换到 zookeeper-3.4.5 目录下，如图 7-43 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt            src
build.xml    docs             NOTICE.txt             zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt   zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt             zookeeper-3.4.5.jar.md5
contrib      lib              recipes                zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]#
```

图 7-43　切换目录

（2）执行命令 bin/zkServer.sh start，启动 Zookeeper 服务进程,如图 7-44 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]#
```

图 7-44　启动服务

（3）执行命令 bin/zkServer.sh status，查看服务是否启动,如图 7-45 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh status
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: standalone
[root@host01 zookeeper-3.4.5]#
```

图 7-45　查看服务状态

（4）执行命令 bin/zkCli.sh -timeout 5000 -server 192.168.1.132:2181，连接测试 Zookeeper 服务，如图 7-46 所示。

```
Mode: standalone
[root@host01 zookeeper-3.4.5]# ./bin/zkCli.sh -timeout 5000 -server 192.168.0.201:2181
Connecting to 192.168.0.201:2181
2016-07-13 13:59:41,733 [myid:] - INFO  [main:Environment@100] - Client environment:zook
eeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:host
.name=host01
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:java
.version=1.7.0_79
2016-07-13 13:59:41,738 [myid:] - INFO  [main:Environment@100] - Client environment:java
.vendor=Oracle Corporation
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.home=/simple/jdk1.7.0_79/jre
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.class.path=/simple/zookeeper-3.4.5/bin/../build/classes:/simple/zookeeper-3.4.5/bin/../
build/lib/*.jar:/simple/zookeeper-3.4.5/bin/../lib/slf4j-log4j12-1.6.1.jar:/simple/zooke
eper-3.4.5/bin/../lib/slf4j-api-1.6.1.jar:/simple/zookeeper-3.4.5/bin/../lib/netty-3.2.2
.Final.jar:/simple/zookeeper-3.4.5/bin/../lib/log4j-1.2.15.jar:/simple/zookeeper-3.4.5/b
in/../lib/jline-0.9.94.jar:/simple/zookeeper-3.4.5/bin/../zookeeper-3.4.5.jar:/simple/zo
okeeper-3.4.5/bin/../src/java/lib/*.jar:/simple/zookeeper-3.4.5/bin/../conf:
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.library.path=/usr/java/packages/lib/i386:/lib:/usr/lib
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.io.tmpdir=/tmp
```

图 7-46　连接测试

（5）在当前项目 zkTest 的 com.zktest 包中，单击右键，选择“New”→“Class”创建“UpdateNodeSync”类，如图 7-47 所示。

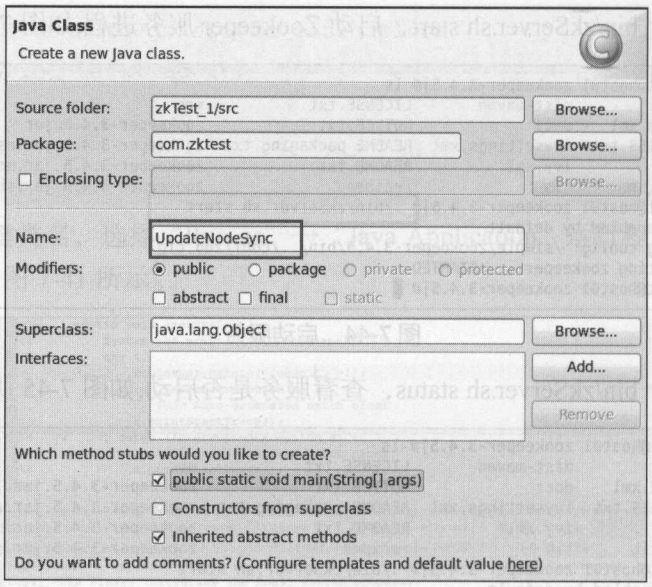



图 7-47　类名

（6）在项目上通过单击右键，选择“Build Path”→“Configure Build Path”→ “Libraries” → “add External JARs”，选择目录文件到/simple/zookeeper-3.4.5/lib 下的 jar 包，添加相关 jar 即可（共 6 个 jar），如图 7-48 所示。

注：分别添加 zookeeper.jar 和 lib 目录下的所有 jar 包。

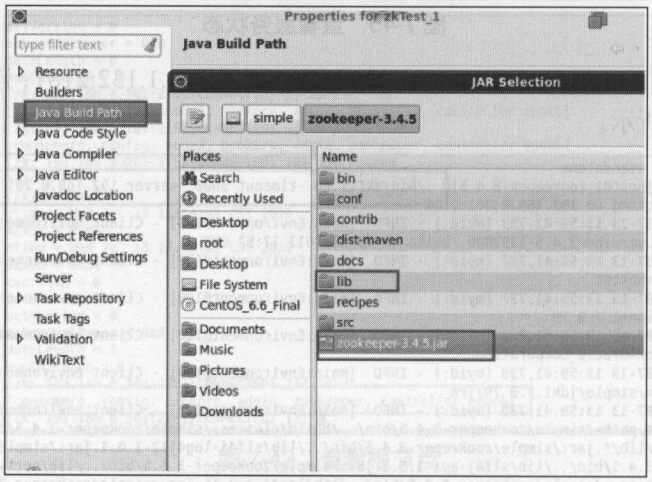



图 7-48　导入 jar 包

（7）在类中实现 Watcher 接口并实现其方法“process(WatchedEvent event)”，如下所示。

```
public void process(WatchedEvent event) {
}
```

（8）利用 zookeeper 对象创建会话并设置连接 zookeeper 后休眠，如下所示。

```
zookeeper = new ZooKeeper("192.168.1.132:2181", 5000, new UpdateNodeSync());
Thread.sleep(Integer.MAX_VALUE);
```

（9）在“process(WatchedEvent event)”中加入判断当前状态及事件类型，如下所示。

```
if(event.getState() == KeeperState.SyncConnected){
        if(event.getType() == EventType.None&& null == event.getPath()){
        }
}
```

（10）创建“public void executeZk()”方法，并在其中利用 zookeeper 对象调用修改节点方法。如果不存在 node3，执行命令 create /node3　12345，代码如下所示。

```
public void executeZk(){
        System.out.println("执行 Zookeeper 操作");
        try {
            Stat stat = zookeeper.setData("/node3","123456".getBytes(), -1);
            System.out.println(stat);
        } catch (KeeperException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

（11）在“process(WatchedEvent event)”方法中判断如果服务连接成功并且事件类型返回 None 则调用“executeZk()”方法，代码如下所示。

```
if(event.getState() == KeeperState.SyncConnected){
        if(event.getType() == EventType.None&& null == event.getPath()){
                executeZk();
        }
}
```

（12）以上程序完整代码如下所示。

```
package com.zktest;
public class UpdateNodeSync implements Watcher{
    public static ZooKeeper zookeeper = null;
    public static void main(String[] args)  throws Exception{
        zookeeper   =   new   ZooKeeper("192.168.0.201:2181",   5000,   new
UpdateNodeSync());
        Thread.sleep(Integer.MAX_VALUE);
    }
    @Override
    public void process(WatchedEvent event) {
        if(event.getState() == KeeperState.SyncConnected){
```

```
            if(event.getType() == EventType.None&& null == event.getPath()){
                executeZk();
            }
        }
    }
    public void executeZk(){
        System.out.println("执行 Zookeeper 操作");
        try {
            Stat stat = zookeeper.setData("/node3", "123456".getBytes(), -1);
            System.out.println(stat);
        } catch (KeeperException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

（13）选中该类文件，单击右键，选择“Run as”→“Java Application”运行程序，返回同步修改节点路径，如图 7-49 所示。

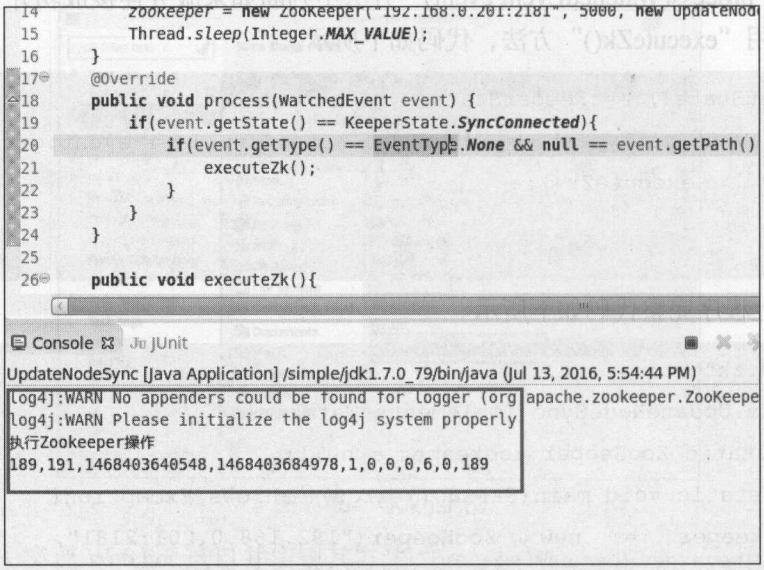



图 7-49　控制台信息

（14）进入 Linux 服务器并启动命令终端，通过运行 Zookeeper 客户端脚本 zkCli.sh 连接服务，并查看节点的值是否已为修改后的值，如图 7-50 所示。

【案例 7-6】Zookeeper Java API——获取子节点（同步）

（1）进入 Linux 服务器，通过命令终端切换到/simple 目录下，然后切换到 zookeeper

-3.4.5 目录下，执行命令 ls，查看目录文件，如图 7-51 所示。

```
mtime = Wed Jul 13 17:54:44 CST 2016
pZxid = 0xbd
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
[zk: 192.168.0.201:2181(CONNECTED) 22] ls /
[consumers, config, node3, brokers, admin, zookeeper, controller_epoch]
[zk: 192.168.0.201:2181(CONNECTED) 23] get /node3
123456
cZxid = 0xbd
ctime = Wed Jul 13 17:54:00 CST 2016
mZxid = 0xbf
mtime = Wed Jul 13 17:54:44 CST 2016
pZxid = 0xbd
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
[zk: 192.168.0.201:2181(CONNECTED) 24]
```

图 7-50　查看根节点值

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]#
```

图 7-51　切换目录

（2）执行命令 bin/zkServer.sh start，启动 Zookeeper 服务进程，如图 7-52 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]#
```

图 7-52　启动服务

（3）执行命令 bin/zkServer.sh status，查看服务是否启动，如图 7-53 所示。

```
[root@host01 zookeeper-3.4.5]# ls
bin          dist-maven       LICENSE.txt          src
build.xml    docs             NOTICE.txt           zookeeper-3.4.5.jar
CHANGES.txt  ivysettings.xml  README_packaging.txt  zookeeper-3.4.5.jar.asc
conf         ivy.xml          README.txt           zookeeper-3.4.5.jar.md5
contrib      lib              recipes              zookeeper-3.4.5.jar.sha1
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@host01 zookeeper-3.4.5]# ./bin/zkServer.sh status
JMX enabled by default
Using config: /simple/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: standalone
[root@host01 zookeeper-3.4.5]#
```

图 7-53　查看服务状态

（4）执行命令 bin/zkCli.sh -timeout 5000 -server 192.168.1.132:2181，连接测试 Zookeeper 服务，如图 7-54 所示。

```
Mode: standalone
[root@host01 zookeeper-3.4.5]# ./bin/zkCli.sh -timeout 5000 -server 192.168.0.201:2181
Connecting to 192.168.0.201:2181
2016-07-13 13:59:41,733 [myid:] - INFO  [main:Environment@100] - Client environment:zook
eeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:host
.name=host01
2016-07-13 13:59:41,737 [myid:] - INFO  [main:Environment@100] - Client environment:java
.version=1.7.0_79
2016-07-13 13:59:41,738 [myid:] - INFO  [main:Environment@100] - Client environment:java
.vendor=Oracle Corporation
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.home=/simple/jdk1.7.0_79/jre
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.class.path=/simple/zookeeper-3.4.5/bin/../build/classes:/simple/zookeeper-3.4.5/bin/../
build/lib/*.jar:/simple/zookeeper-3.4.5/bin/../lib/slf4j-log4j12-1.6.1.jar:/simple/zooke
eper-3.4.5/bin/../lib/slf4j-api-1.6.1.jar:/simple/zookeeper-3.4.5/bin/../lib/netty-3.2.2
.Final.jar:/simple/zookeeper-3.4.5/bin/../lib/log4j-1.2.15.jar:/simple/zookeeper-3.4.5/b
in/../lib/jline-0.9.94.jar:/simple/zookeeper-3.4.5/bin/../zookeeper-3.4.5.jar:/simple/zo
okeeper-3.4.5/bin/../src/java/lib/*.jar:/simple/zookeeper-3.4.5/bin/../conf:
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.library.path=/usr/java/packages/lib/i386:/lib:/usr/lib
2016-07-13 13:59:41,739 [myid:] - INFO  [main:Environment@100] - Client environment:java
.io.tmpdir=/tmp
```

图 7-54　连接 Zookeeper 服务

（5）在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”，新建一个项目“zkTest_1”，如图 7-55 所示。

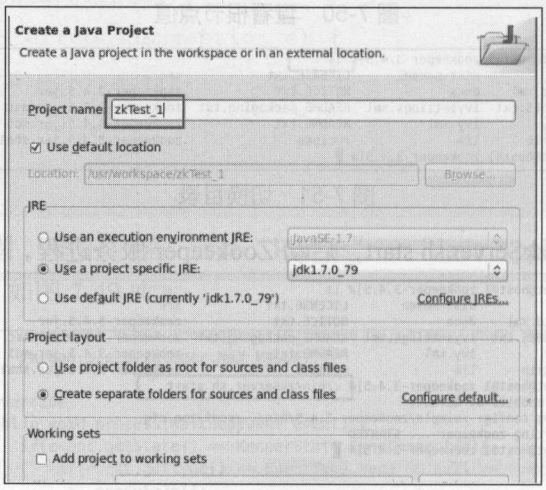



图 7-55　新建项目

（6）在项目上通过单击右键，选择“Build Path”→“Configure Build Path”→“Libraries”→ “add External JARs”，选择目录文件到/simple/zookeeper-3.4.5/lib 下的 jar 包，添加相关 jar 即可（共 6 个 jar），如图 7-56 所示。

注：分别添加 zookeeper.jar 和 lib 目录下的所有 jar 包。

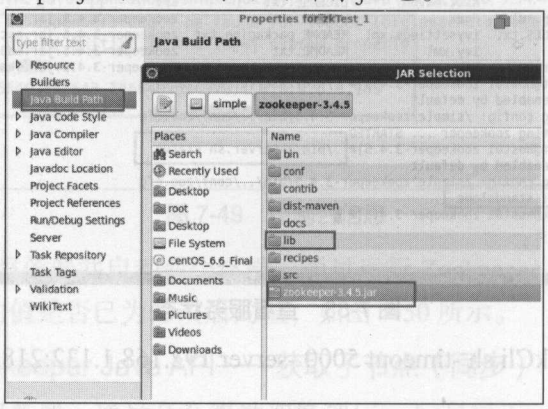



图 7-56　导入 jar 包

（7）在当前项目 zkTest_1 的“com.zktest”包下，单击右键，选择“New”→“Class”创建“GetChildrenNodeSync”类，如图 7-57 所示。

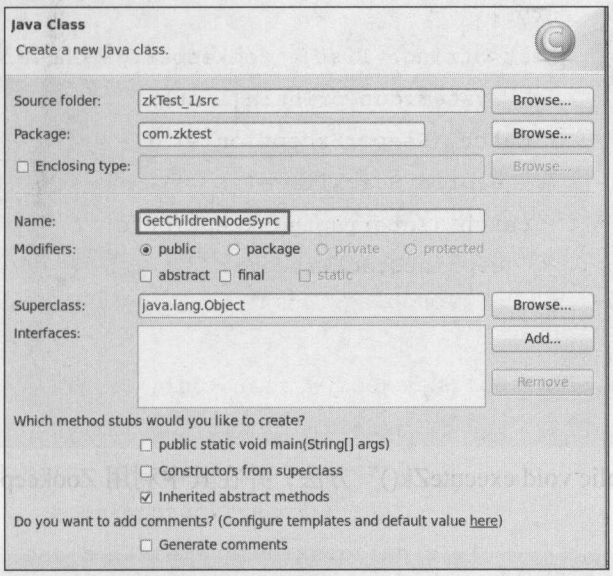



图 7-57 类名

（8）将类实现 Watcher 接口并实现其方法“process(WatchedEvent event)”，如下所示。

```
public class CreateNodeASync implements Watcher{
    public void process(WatchedEvent event) {
    }
}
```

（9）利用 Zookeeper 对象创建会话并设置连接 Zookeeper 后休眠，如下所示。

```
zookeeper = new ZooKeeper("192.168.1.132:2181", 5000, new CreateNodeASync());
          Thread.sleep(Integer.MAX_VALUE);
```

（10）在“process(WatchedEvent event)”中加入判断当前状态及事件类型，如下所示。

```
        // TODO Auto-generated method stub
        System.out.println("回调事件类型: "+event.getType());
        if(event.getState() == KeeperState.SyncConnected){
            if(event.getType() == EventType.None&& null == event.getPath()){
                try {
                    List<String> list = zookeeper.getChildren("/", true);
                    System.out.println(list);
                } catch (KeeperException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
        }else{
            if(event.getType() == EventType.NodeChildrenChanged){
                try {
                List<String> list = zookeeper.getChildren("/", true);
                    System.out.println(list);
                } catch (KeeperException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

（11）创建“public void executeZk()”方法，并在其中利用 Zookeeper 对象获取子节点，代码如下所示。

```
public void executeZk(){
        try {
            List<String> list = zookeeper.getChildren("/",true);
            System.out.println(list);
        } catch (KeeperException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

（12）在“process(WatchedEvent event)”方法中判断如果服务连接成功并且事件类型返回 None 则调用“executeZk()”方法，代码如下所示。

```
if(event.getState() == KeeperState.SyncConnected){
        if(event.getType() == EventType.None&& null == event.getPath()){
                executeZk();
        }
}
```

（13）以上完整代码如下所示。

```
package com.zktest;
public class GetChildrenNodeSync implements Watcher{
    private static ZooKeeper zookeeper;
    public   static   void   main(String[]   args)   throws   IOException,
```

```
InterruptedException {
        zookeeper = new ZooKeeper("192.168.0.201:2181", 5000, new
GetChildrenNodeSync())
        Thread.sleep(Integer.MAX_VALUE);
    }
    @Override
    public void process(WatchedEvent event) {
        System.out.println("回调事件类型: "+event.getType());
        if(event.getState() == KeeperState.SyncConnected){
            if(event.getType() == EventType.None&& null == event.getPath()){
                try {
                    List<String> list = zookeeper.getChildren("/", true);
                    System.out.println(list);
                } catch (KeeperException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }else{
                if(event.getType() == EventType.NodeChildrenChanged){
                    try {
                        List<String> list = zookeeper.getChildren("/", true);
                        System.out.println(list);
                    } catch (KeeperException e) {
                        e.printStackTrace();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

（14）选中 Java 程序，单击右键，选择“Run as”→“Java Application”运行程序，返回同步获取子节点列表，结果如图 7-58 所示。

（15）进入 Linux 服务器并启动命令终端，运行 Zookeeper 的脚本 zkCli.sh 连接服务，并查看“/”下节点和控制台打印的子节点列表是否一致，如图 7-59 所示。

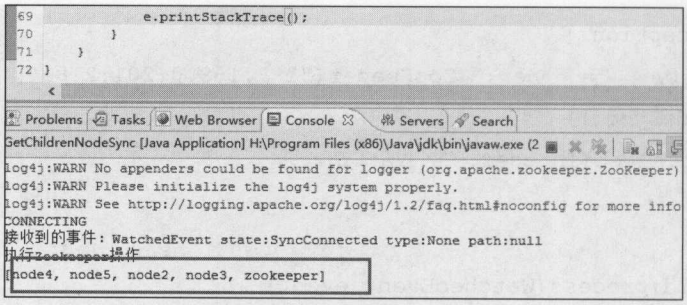



图 7-58　控制台信息

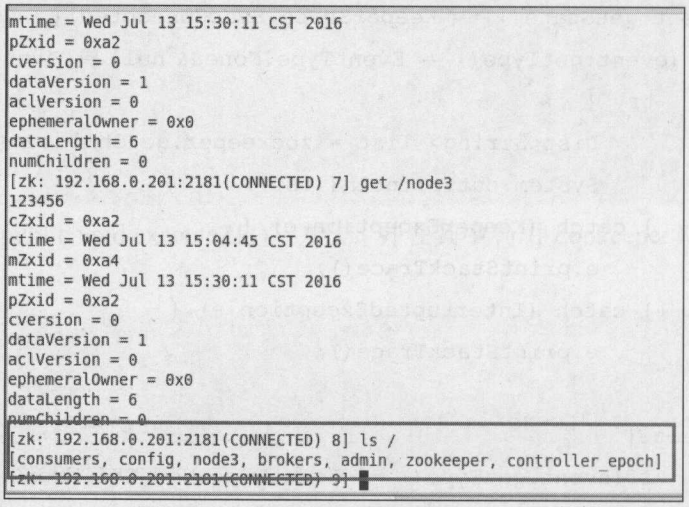



图 7-59　查看子节点

## 本章小结

在本章中，首先介绍了 Zookeeper 的概念，然后介绍了 Zookeeper 术语，便于读写对 Zookeeper 工作原理的理解，最后对 Zookeeper 事件和 Zookeeper 操作进行讲解，最终达到应用于现实的工作当中，解决分布式应用的协调一致性问题。

## 习题

1. 简述 Zookeeper 的选举机制。
2. 简述 Zookeeper 的监听机制。
3. 简述 Zookeeper 系统的意义。

扫一扫在线测

# 第 8 章 Hadoop 数据库 Hbase

## 本章要点

- Hbase 简介。
- Hbase 优势和特点。
- Hbase 架构体系。
- Hbase Shell 命令。
- Hbase API 操作。

## 引　言

Hbase 是一个分布式的，面向列的开源数据库，可以称为 Hadoop 的标准数据库，也是一款比较流行的 NoSQL 数据库，由 Google 发表的论文 Bigtable 经过演变而来。Hbase 在 Hadoop 之上提供了类似 Bigtable 的能力，主要解决非关系型数据存储问题。本章通过对 Hbase 的概述、Hbase 的架构、Hbase Shell 命令、Hbase API 操作的讲解，让学生深刻理解并学会运用 Hbase 系统。

## 8.1 Hbase 概述

### 8.1.1 Hbase 简介

官方网址：http://hbase.apache.org/。

官方文档：http://abloz.com/hbase/book.html。

（1）Hbase 是一个可分布、可扩展的大数据存储的 Hadoop 数据库。适用于随机、实时读写大数据操作时使用。它的目标就是拥有一张大表，支持亿行亿列。Hbase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

（2）Hbase 是一个分布式的、面向列的开源数据库,源于 Google 的一篇论文《bigtable：一个结构化数据的分布式存储系统》。Hbase 是 Google Bigtable 的开源实现，它利用 Hadoop HDFS 作为其文件存储系统，利用 Hadoop MapReduce 来处理 HBase 中的海量数据，利用 Zookeeper 作为协同服务。

（3）Hbase – Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。Hbase 利用 Hadoop HDFS 作为其文件存储系统，利用 Hadoop MapReduce 来处理 Hbase 中的海量数据，利用 Zookeeper 作为协调工具。

（4）Hbase 表的数据可以存储在本地，也可以存储在 HDFS 之上。

### 8.1.2 Hbase 优势和特点

Hbase 的优势如下。

- 成熟：社区成熟，支持工具丰富。
- 高效：适应高并发写入。
- 分布式：基于 HDFS，易扩展。

Hbase 的特点如下。

- 一个大表，亿行万列。
- 面向列：面向列族存储和权限访问，列和列族都是独立索引。
- 对于为 null 的列是不占用存储空间，所以表的设计很稀疏。
- 数据类型单一，都是字符串类型。
- 同一行可以有截然不同的列。
- 介于关系型和非关系型数据库之间，实现非结构化和半结构化的数据存储。

### 8.1.3 Hbase 专业术语

#### 1. 行键（Row Key）

Row key（行键）可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10～100bytes)，在 Hbase 内部，Row Key 保存为字节数组。存储时，数据按照 Row Key 的字典序(byte order)排序存储。设计 key 时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。

#### 2. 单元（Cell）

Hbase 中通过 row 和 columns 确定的为一个存储单元，称为 cell，是由{row key, column, version} 唯一确定的单元。cell 中的数据是没有类型的，全部是以字节码存储。在写入数据的时候，时间戳是由系统时间自动赋值，精确到毫秒，也可以显示的赋值。在一个 cell 中同一数据的不同版本的顺序是按照时间的倒序排序的。

#### 3. 列族

列族是在创建表时声明，一个列族可以包含多个列，列中的数据都是以二进制形式存在的，没有数据类型。列族是由多个列而形成的集合。一个列族中的所有的列成员有着相同的前缀。比如 course：math，冒号是列族名和列名的分隔符。

#### 4. 时间戳

Hbase 中通过 row 和 columns 确定的为一个存储单元，称为 cell。每个 cell 都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 Hbase(在数据写入时自动 )赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。

## 8.2 Hbase 架构

Hbase 采用 Master/Slave 架构搭建集群，它隶属于 Hadoop 生态系统，由以下类型节点

组成：HMaster 节点、HRegionServer 节点、Zookeeper 集群，而在底层，它将数据存储于 HDFS 中，因而涉及 HDFS 的 NameNode、DataNode，如图 8-1 所示。

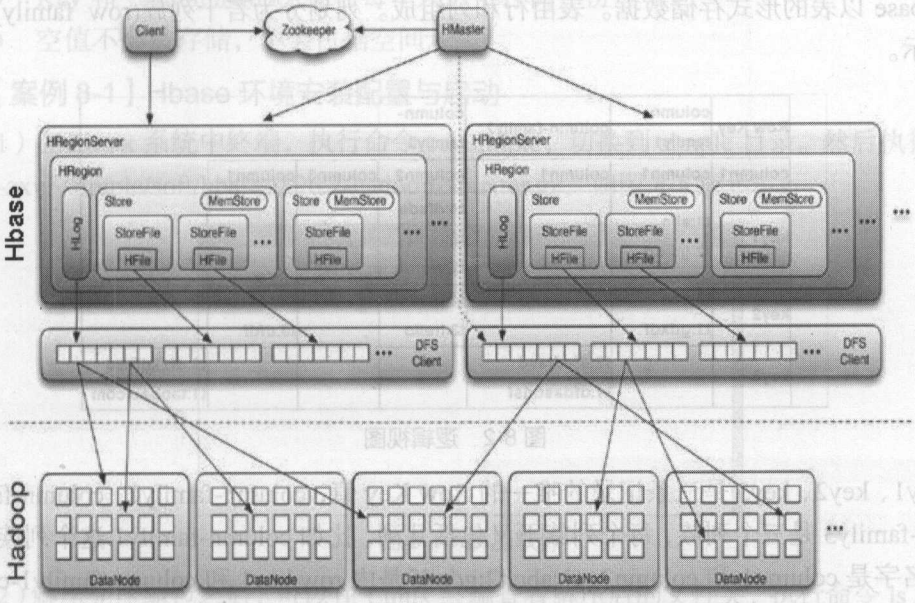



图 8-1　Hbase 架构

## 8.2.1　角色

### 1. Client

包含访问 Hbase 的接口，Client 维护着一些 cache 来加快对 Hbase 的访问，比如 Region 的位置信息。

### 2. Zookeeper

- 保证任何时候，集群中只有一个 running master。
- 存储所有 Region 的寻址入口。
- 实时监控 Region Server 的状态，将 Region Server 的上线和下线信息，实时通知给 Master。
- 存储 Hbase 的 schema,包括有哪些 table，每个 table 有哪些 column family。

### 3. Master 可以启动多个 HMaster

通过 Zookeeper 的 Master Election 机制保证总有一个 Master 运行。

- 为 Region Server 分配 Region。
- 负责 Region Server 的负载均衡。
- 发现失效的 Region Server 并重新分配其上的 Region。

### 4. Region Server

维护 Master 分配给它的 Region，处理对这些 Region 的 IO 请求。负责切分在运行过程中变得过大的 Region。

### 8.2.2 Hbase 物理存储和逻辑视图

（1）逻辑视图。

Hbase 以表的形式存储数据。表由行和列组成。列划分为若干列族(row family)，如图 8-2 所示。

| Row Key | column-family1 | column-family2 | column-family3 | | | |
|---|---|---|---|---|---|---|
| column1 | column1 | column1 | column2 | column3 | column1 | |
| key1 | t1:abc<br>t2:gdxdf | | t4:dfads<br>t3:hello<br>t2:world | | | |
| key2 | t3:abc<br>t1:gdxdf | | t4:dfads<br>t3:hello | | t2:dfdsfa<br>t3:dfdf | |
| key3 | | t2:dfadfasd<br>t1:dfdasddsf | | | | t2:dfxxdfasd<br>t1:taobao.com |

图 8-2 逻辑视图

key1、key2、key3 是三条记录的唯一的 Row Key 值，column-family1，column-family2，column-family3 是三个列族，每个列族下又包括几列。比如 column-family1 这个列族下包括两列，名字是 column1 和 column2，t1:abc,t2:gdxdf 是由 row key1 和 column-family1-column1 唯一确定的一个单元 cell。这个 cell 中有两个数据，abc 和 gdxdf。两个值的时间戳不一样，分别是 t1,t2, hbase 会返回最新时间的值给请求者。

（2）物理存储。

Hbase 的 Table 中的所有行都按照 Row Key 的字典序排列。Table 在行的方向上分割为多个 HRegion。Region 按大小分割的，每个表一开始只有一个 Region，随着数据不断插入表，Region 不断增大，当增大到一个阀值的时候，HRegion 就会等分为两个新的 HRegion。当 Table 中的行不断增多，就会有越来越多的 HRegion。HRegion 是 Hbase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 HRegion 可以分布在不同的 HRegion server 上。但一个 HRegion 是不会拆分到多个 Server 上的。如图 8-3 所示。

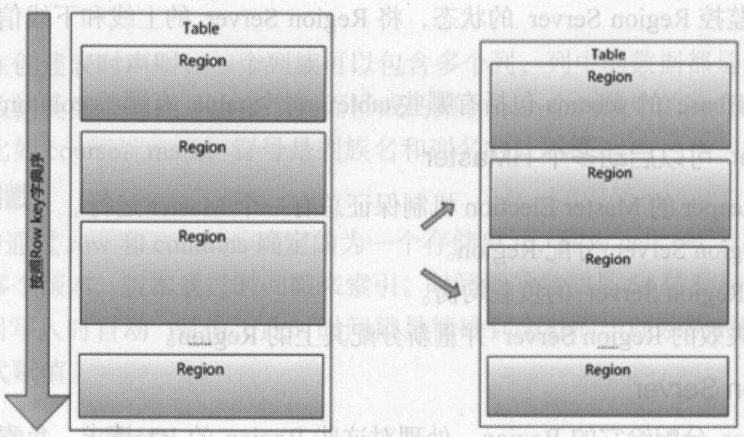



图 8-3 物理存储

Hbase 的存储细节：

- 每个列族存储在 HDFS 上的一个单独文件夹中。
- Key 和 version number 会在每个列族中存储一份。
- 空值不会被存储，不会占据空间。

**【案例 8-1】Hbase 环境安装配置与启动**

（1）在 Linux 系统中终端，执行命令 cd /simple，切换到 simple 目录。然后执行解压命令 tar -zxvf /simple/soft/hbase-0.96.2-hadoop2-bin.tar.gz，如图 8-4 所示。

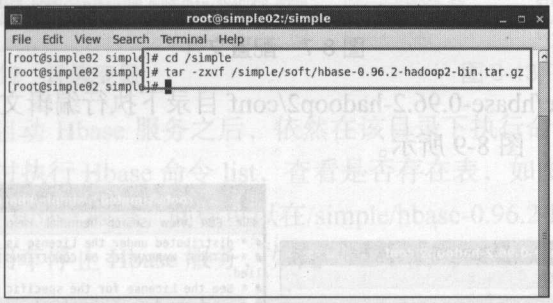



图 8-4　解压软件

（2）解压完压缩包之后，可以在 Linux 终端查看解压后的文件夹，执行命令 ls /simple/，如图 8-5 所示。

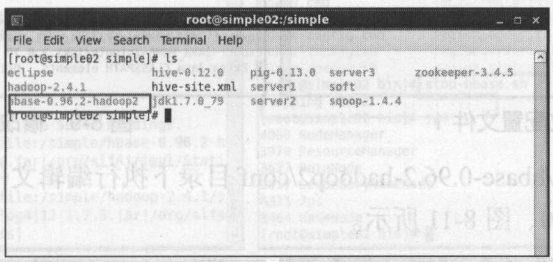



图 8-5　查看解压后的软件

（3）在 Linux 终端输入切换命令 cd /simple/hbase-0.96.2-hadoop2 进入到 Hbase 安装目录并查看目录结构，如图 8-6 所示。

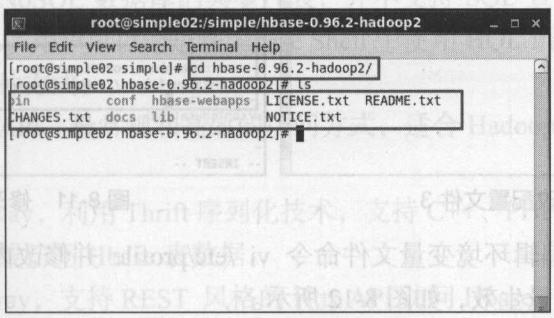



图 8-6　Hbase 安装目录

（4）在终端通过执行切换命令 cd conf 进入 Hbase 配置目录并查看配置文件，如图 8-7 所示。

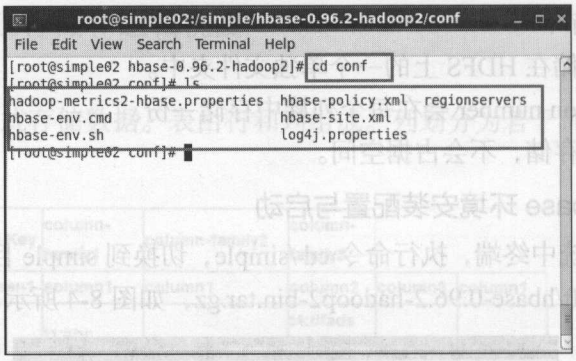



图 8-7　配置文件

（5）在终端/simple/hbase-0.96.2-hadoop2/conf 目录下执行编辑文件命令 vi hbase-env.sh 并修改内容，如图 8-8、图 8-9 所示。

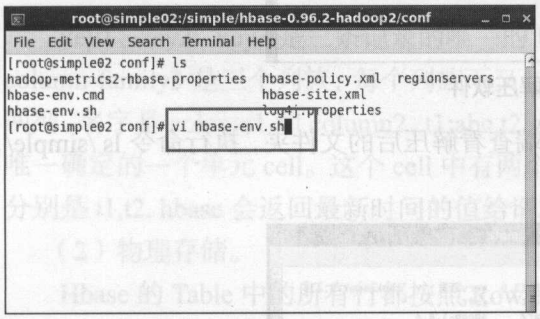



图 8-8　修改配置文件 1

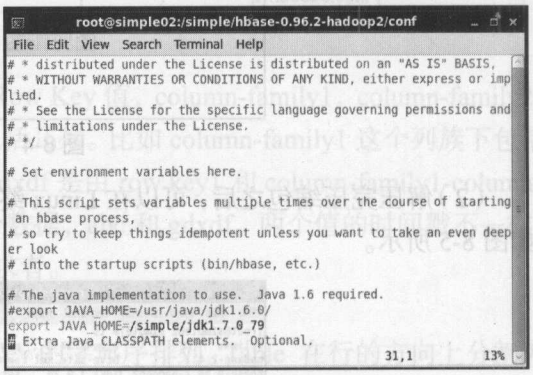



图 8-9　修改配置文件 2

（6）在终端/simple/hbase-0.96.2-hadoop2/conf 目录下执行编辑文件命令 vi hbase-site.xml 并修改内容，如图 8-10、图 8-11 所示。

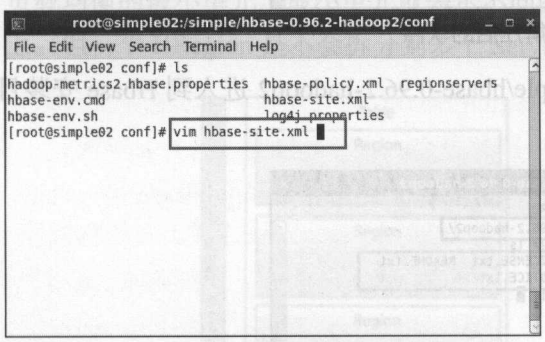



图 8-10　修改配置文件 3

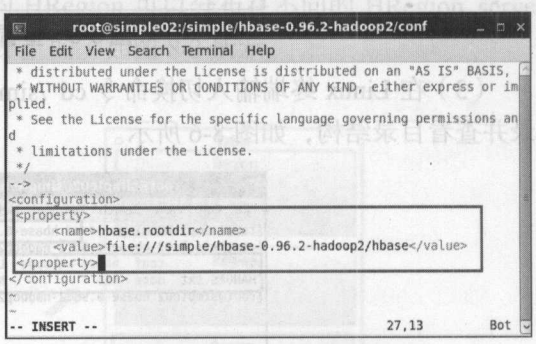



图 8-11　修改配置文件 4

（7）在终端执行编辑环境变量文件命令 vi /etc/profile 并修改内容，执行命令 source /etc/profile 让其环境变量生效，如图 8-12 所示。

（8）在 Linux 系统中终端首先切换到/simple/hbase-0.96.2-hadoop2/bin 目录，执行命令./start-hbase.sh，回车启动 Hbase 服务。然后执行查看进程命令 jps 查看 HMaster 进程是否启动，如图 8-13 所示。

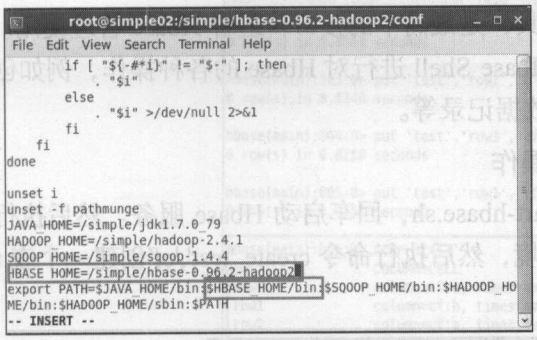



图 8-12 环境变量

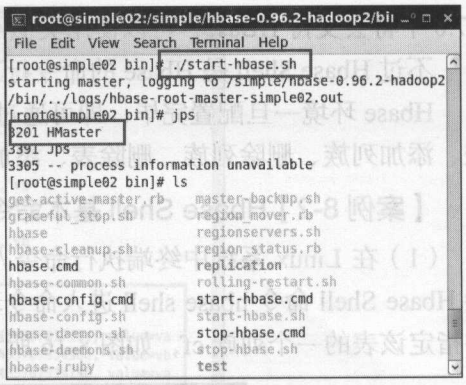



图 8-13 启动 Hbase

（9）上一步操作启动 Hbase 服务之后，依然在该目录下执行命令 hbase shell，进入到 Hbase Shell 环境并通过执行 Hbase 命令 list，查看是否存在表，如图 8-14 所示。

（10）如果想停止 Hbase 服务，同样可以在/simple/hbase-0.96.2-hadoop2/bin 目录，执行命令./stop-hbase.sh，回车停止 Hbase 服务，如图 8-15 所示。

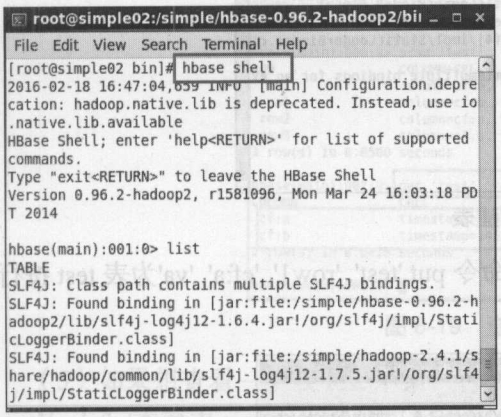



图 8-14 Shell 环境

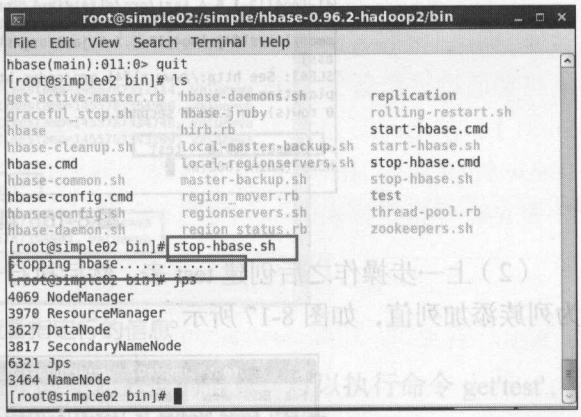



图 8-15 停止 Hbase

## 8.3 Hbase Shell 操作

Hbase 应该属于 NoSQL 数据库的典型代表，并不支持 SQL 查询语句，但是 Hbase 有自带的查询语句 HQL，用户可以通过在 Hbase Shell 中使用 HQL 语句进行数据的查询。当然也可以通过下列 5 种方式：

（1）Native Java API，最常规和高效的访问方式，适合 Hadoop MapReduce Job 并行批处理 Hbase 表数据。

（2）Thrift Gateway，利用 Thrift 序列化技术，支持 C++、PHP、Python 等多种语言，适合其他异构系统在线访问 Hbase 表数据。

（3）REST Gateway，支持 REST 风格的 Http API 访问 Hbase，解除了语言限制。

（4）Pig，可以使用 Pig Latin 流式编程语言来操作 Hbase 中的数据，和 Hive 类似，本质最终也是编译成 MapReduce Job 来处理 HBase 表数据，适合做数据统计。

（5）Hive，当前 Hive 的 Release 版本尚没有加入对 Hbase 的支持，但在下一个版本 Hive

0.7.0 中将会支持 HBase，可以使用类似 SQL 语言来访问 Hbase。

不过 Hbase Shell 是 Hbase 的命令行工具，是最简单的接口，适合 Hbase 管理使用。

Hbase 环境一旦配置完毕，即可通过 Hbase Shell 进行对 Hbase 的各种操作，例如创建表、添加列族、删除列族、删除表、添加数据记录等。

【案例 8-2】Hbase Shell 基本命令操作

（1）在 Linux 系统中终端执行命令./start-hbase.sh，回车启动 Hbase 服务。然后执行连接 Hbase Shell 命令 hbase shell 进入命令环境，然后执行命令 create 'test','cf'创建一个表 test 并指定该表的一个列族 cf，如图 8-16 所示。

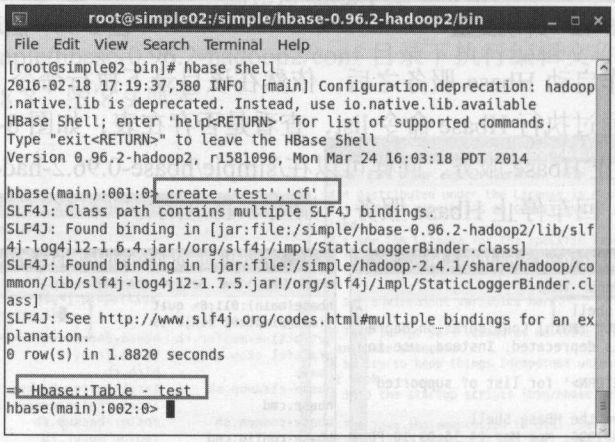



图 8-16　创建表

（2）上一步操作之后创建 test 表，然后执行命令 put 'test', 'row1', 'cf:a', 'va'为表 test 按行为列族添加列值，如图 8-17 所示。

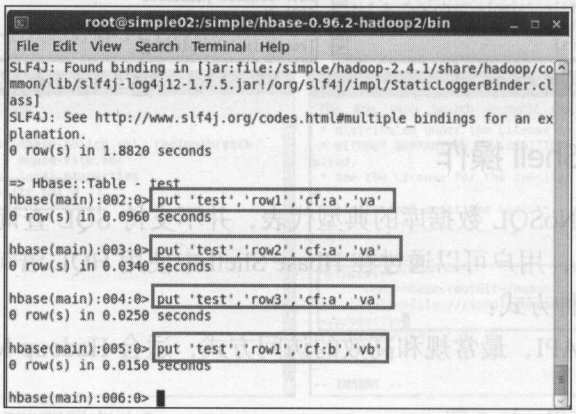



图 8-17　添加值

（3）如果想查看一下表 test 中添加的行信息，可以执行命令 scan'test'，查看表 test 中所有的信息，如图 8-18 所示。

（4）如果想查看一下表 test 中某一行的信息，可以执行命令 get 'test'，'row1'查看表 test 中指定行 row 的所有的信息，如图 8-19 所示。

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help
hbase(main):002:0> put 'test','row1','cf:a','va'
0 row(s) in 0.0960 seconds

hbase(main):003:0> put 'test','row2','cf:a','va'
0 row(s) in 0.0340 seconds

hbase(main):004:0> put 'test','row3','cf:a','va'
0 row(s) in 0.0250 seconds

hbase(main):005:0> put 'test','row1','cf:b','vb'
0 row(s) in 0.0150 seconds

hbase(main):006:0> scan 'test'
ROW                 COLUMN+CELL
 row1               column=cf:a, timestamp=1455787317114, value=va
 row1               column=cf:b, timestamp=1455787354908, value=vb
 row2               column=cf:a, timestamp=1455787334748, value=va
 row3               column=cf:a, timestamp=1455787343209, value=va
3 row(s) in 0.0580 seconds

hbase(main):007:0>
```

图 8-18　查看表

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help
hbase(main):004:0> put 'test','row3','cf:a','va'
0 row(s) in 0.0250 seconds

hbase(main):005:0> put 'test','row1','cf:b','vb'
0 row(s) in 0.0150 seconds

hbase(main):006:0> scan 'test'
ROW                 COLUMN+CELL
 row1               column=cf:a, timestamp=1455787317114, value=va
 row1               column=cf:b, timestamp=1455787354908, value=vb
 row2               column=cf:a, timestamp=1455787334748, value=va
 row3               column=cf:a, timestamp=1455787343209, value=va
3 row(s) in 0.0580 seconds

hbase(main):007:0> get 'test','row1'
COLUMN              CELL
 cf:a               timestamp=1455787317114, value=va
 cf:b               timestamp=1455787354908, value=vb
2 row(s) in 0.0410 seconds

hbase(main):008:0>
```

图 8-19　查看指定行的信息

（5）如果想查看一下表 test 中某一行列族的某列数据的信息，可以执行命令 get'test', 'row1', 'cf:a'查看表 test 中指定某行列族的某列的信息，如图 8-20 所示。

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help
hbase(main):006:0> scan 'test'
ROW                 COLUMN+CELL
 row1               column=cf:a, timestamp=1455787317114, value=va
 row1               column=cf:b, timestamp=1455787354908, value=vb
 row2               column=cf:a, timestamp=1455787334748, value=va
 row3               column=cf:a, timestamp=1455787343209, value=va
3 row(s) in 0.0580 seconds

hbase(main):007:0> get 'test','row1'
COLUMN              CELL
 cf:a               timestamp=1455787317114, value=va
 cf:b               timestamp=1455787354908, value=vb
2 row(s) in 0.0410 seconds

hbase(main):008:0> get 'test','row1','cf:a'
COLUMN              CELL
 cf:a               timestamp=1455787317114, value=va
1 row(s) in 0.0710 seconds

hbase(main):009:0>
```

图 8-20　查看指定列信息

（6）如果想删除前面创建的表 test，需要首先执行命令 disable'test'让该表处于无效状态，然后删除该表。可以执行命令 drop 'test'删除该表，如图 8-21 所示。

（7）在 Hbase Shell 环境下执行相关操作完毕之后，想退出该命令环境，可以执行命令 quit 即可退出，如图 8-22 所示。

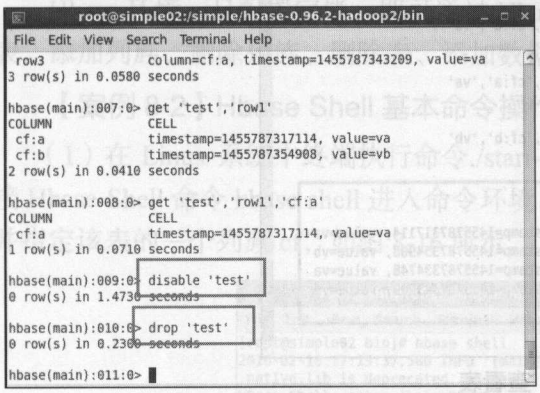



图 8-21　删除表

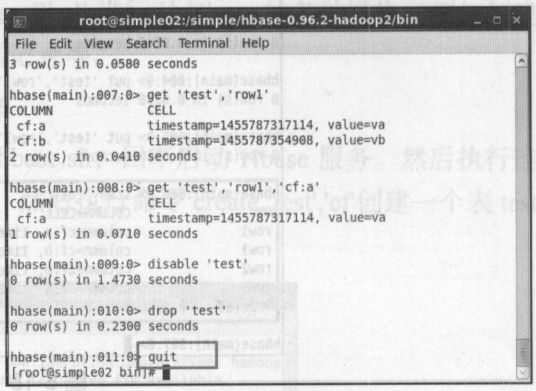



图 8-22　退出 Shell 环境

【案例 8-3】Hbase Shell 操作用户表

Hbase 是一个分布式的、面向列的开源数据库，它利用 Hadoop HDFS 作为其文件存储系统，利用 Hadoop MapReduce 来处理 Hbase 中的海量数据。本试验 Hbase 提供了一个 Shell 的终端通过操作命令对表 user 操作。

（1）在 Linux 系统中终端执行命令./start-hbase.sh，回车启动 Hbase 服务。执行连接 Hbase Shell 命令 Hbase Shell 进入命令环境，然后执行命令 create 'user', 'info', 'data'，创建一个表 user 并指定该表的一个列族 info 和 data，如图 8-23 所示。

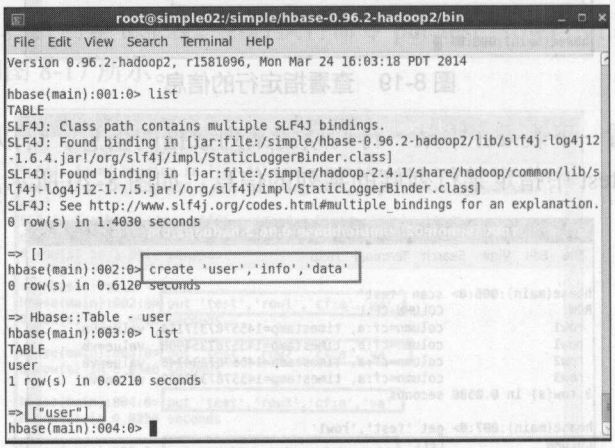



图 8-23　创建表

（2）创建 user 表后需要向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 name 列标示符，值为 zhangsan，执行命令 put 'user', 'rk0001', 'info:name', 'zhangsan'。向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 gender 列标示符，值为 female，执行命令 put 'user', 'rk0001', 'info:gender', 'female'。向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 age 列标示符，值为 20。执行命令 put 'user', 'rk0001', 'info:age', 20。如图 8-24 所示。

（3）创建 user 表之后，向 user 表中插入信息，row key 为 rk0001，列族 data 中添加 pic

列标示符，值为 picture，执行命令 put'user', 'rk0001', 'data:pic', 'picture'，如图 8-25 所示。

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help
Here is some help for this command:
Put a cell 'value' at specified table/row/column and optionally
timestamp coordinates.  To put a cell value into table 'ns1:t1' or 't1'
at row 'r1' under column 'c1' marked with the time 'ts1', do:

  hbase> put 'ns1:t1', 'r1', 'c1', 'value', ts1

The same commands also can be run on a table reference. Suppose you had a refere
nce
t to table 't1', the corresponding command would be:

  hbase> t.put 'r1', 'c1', 'value', ts1


hbase(main):005:0> put 'user','rk0001','info:name','zhangsan'
0 row(s) in 0.0960 seconds

hbase(main):006:0> put 'user','rk0001','info:gender','female'
0 row(s) in 0.0070 seconds

hbase(main):007:0> put 'user','rk0001','info:age',20
0 row(s) in 0.0170 seconds

hbase(main):008:0>
```

图 8-24　插入数据 1

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help

The same commands also can be run on a table reference. Suppose you had a refere
nce
t to table 't1', the corresponding command would be:

  hbase> t.put 'r1', 'c1', 'value', ts1


hbase(main):005:0> put 'user','rk0001','info:name','zhangsan'
0 row(s) in 0.0960 seconds

hbase(main):006:0> put 'user','rk0001','info:gender','female'
0 row(s) in 0.0070 seconds

hbase(main):007:0> put 'user','rk0001','info:age',20
0 row(s) in 0.0170 seconds

hbase(main):008:0> put 'user','rk0001','data:pic','picture'
0 row(s) in 0.0240 seconds

hbase(main):009:0> put 'user','rk0001','data:size','30'
0 row(s) in 0.0320 seconds

hbase(main):010:0>
```

图 8-25　插入数据 2

（4）查看 user 表中相关数据。获取 user 表中 row key 为 rk0001 的所有信息，执行命令 get'user', 'rk0001'。获取 user 表中 row key 为 rk0001，info 列族的所有信息，执行命令 get'user', 'rk0001', 'info'。获取 user 表中 row key 为 rk0001，info 列族的 name、age 列标示符的信息，执行命令 get'user', 'rk0001', 'info:name', 'info:age'，如图 8-26 所示。

```
root@simple02:/simple/hbase-0.96.2-hadoop2/bin
File  Edit  View  Search  Terminal  Help
hbase(main):011:0> get 'user','rk0001'
COLUMN                CELL
 data:pic             timestamp=1455861465083, value=picture
 data:size            timestamp=1455861479909, value=30
 info:age             timestamp=1455861306837, value=20
 info:gender          timestamp=1455861287866, value=female
 info:name            timestamp=1455861261496, value=zhangsan
5 row(s) in 0.0470 seconds

hbase(main):012:0> get 'user','rk0001','info'
COLUMN                CELL
 info:age             timestamp=1455861306837, value=20
 info:gender          timestamp=1455861287866, value=female
 info:name            timestamp=1455861261496, value=zhangsan
3 row(s) in 0.0220 seconds

hbase(main):013:0> get 'user','rk0001','info:name','info:age'
COLUMN                CELL
 info:age             timestamp=1455861306837, value=20
 info:name            timestamp=1455861261496, value=zhangsan
2 row(s) in 0.0220 seconds

hbase(main):014:0>
```

图 8-26　查看信息 1

（5）获取 user 表中 row key 为 rk0001，info、data 列族的信息，可以执行如下各命令实现：

```
get 'user', 'rk0001', 'info', 'data'
get 'user', 'rk0001', {COLUMN => ['info', 'data']}
get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}
```

如图 8-27 所示。

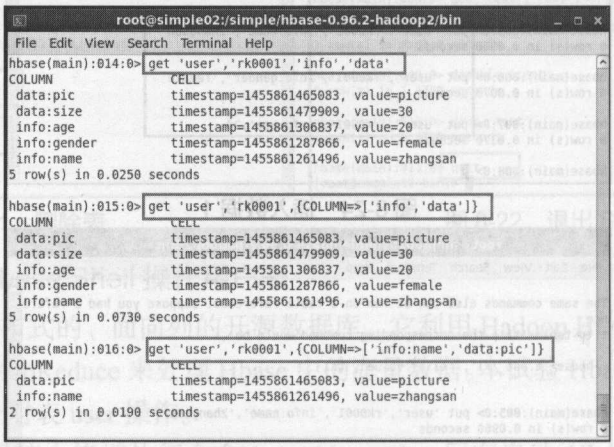



图 8-27　查看信息 2

（6）查看某一行指定列族的列单元中的某个元素。

获取 user 表中 row key 为 rk0001，列族为 info，版本号最新 5 个的信息。

```
get 'user', 'rk0001', {COLUMN => 'info', VERSIONS => 2}
get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5}
```

如图 8-28 所示。

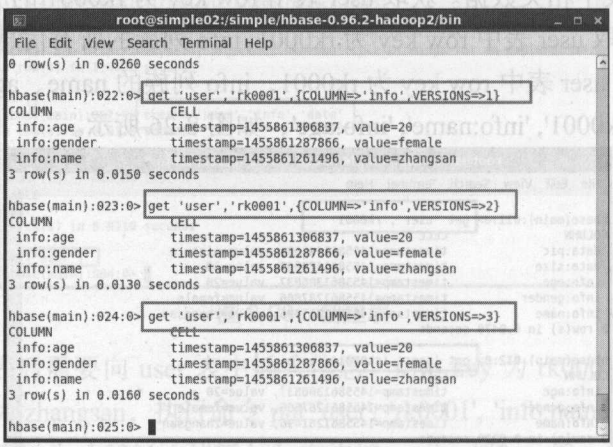



图 8-28　查看信息 3

## 8.4　Hbase API 操作

Hbase API 常用的类有：

（1）HBaseAdmin。

关系：org.apache.hadoop.hbase.client.HBaseAdmin

作用：提供了一个接口来管理 HBase 数据库的表信息。它提供的方法包括创建表、删除表、列出表项、使表有效或无效，以及添加或删除表列族成员等。

| 返回值 | 函数 | 描述 |
| --- | --- | --- |
| void | addColumn(String tableName, HColumnDescriptor column) | 向一个已经存在的表添加列 |
| void | checkHBaseAvailable(HBaseConfiguration conf) | 静态函数，查看 HBase 是否处于运行状态 |
| void | createTable(HTableDescriptor desc) | 创建一个表，同步操作 |
| void | deleteTable(byte[] tableName) | 删除一个已经存在的表 |
| void | enableTable(byte[] tableName) | 使表处于有效状态 |
| void | disableTable(byte[] tableName) | 使表处于无效状态 |
| void | HTableDescriptor[] listTables() | 列出所有用户控件表项 |
| void | modifyTable(byte[] tableName, HTableDescriptor htd) | 修改表的模式，是异步的操作，可能需要花费一定的时间 |
| void | boolean tableExists(String tableName) | 检查表是否存在 |

用法示例：

```
HBaseAdmin admin = new HBaseAdmin(config);
admin.disableTable("tablename")
```

（2）HBaseConfiguration。

关系：org.apache.hadoop.hbase.HBaseConfiguration

作用：对 Hbase 进行配置。

| 返回值 | 函数 | 描述 |
| --- | --- | --- |
| void | addResource(Path file) | 通过给定的路径所指的文件来添加资源 |
| void | clear() | 清空所有已设置的属性 |
| string | get(String name) | 获取属性名对应的值 |
| String | getBoolean(String name, boolean defaultValue) | 获取为 boolean 类型的属性值，如果其属性值类型不为 boolean，则返回默认属性值 |
| void | set(String name, String value) | 通过属性名来设置值 |
| void | setBoolean(String name, boolean value) | 设置 boolean 类型的属性值 |

用法示例：

```
HBaseConfiguration hconfig = new HBaseConfiguration();
hconfig.set("hbase.zookeeper.property.clientPort","2181");
```

（3）HTableDescriptor。

关系：org.apache.hadoop.hbase.HTableDescriptor

作用：包含了表的名字及其对应表的列族。

| 返回值 | 函数 | 描述 |
| --- | --- | --- |
| void | addFamily(HColumnDescriptor) | 添加一个列族 |
| void | HColumnDescriptor removeFamily(byte[] column) | 移除一个列族 |
| byte[] | getName() | 获取表的名字 |
| byte[] | getValue(byte[] key) | 获取属性的值 |
| void | setValue(String key, String value) | 设置属性的值 |

用法示例：

```
HTableDescriptor htd = new HTableDescriptor(table);
htd.addFamily(new HcolumnDescriptor("family"));
```

（4）Put。

关系：org.apache.hadoop.hbase.client.Put

作用：用来对单个行执行添加操作。

| 返回值 | 函数 | 描述 |
| --- | --- | --- |
| Put | add(byte[] family, byte[] qualifier, byte[] value) | 将指定的列和对应的值添加到 Put 实例中 |
| Put | add(byte[] family, byte[] qualifier, long ts, byte[] value) | 将指定的列和对应的值及时间戳添加到 Put 实例中 |
| byte[] | getRow() | 获取 Put 实例的行 |
| RowLock | getRowLock() | 获取 Put 实例的行锁 |
| long | getTimeStamp() | 获取 Put 实例的时间戳 |
| boolean | isEmpty() | 检查 familyMap 是否为空 |
| Put | setTimeStamp(long timeStamp) | 设置 Put 实例的时间戳 |

用法示例：

```
HTable table = new HTable(conf,Bytes.toBytes(tablename));
Put p = new Put(brow);//为指定行创建一个 Put 操作
p.add(family,qualifier,value);
table.put(p);
```

（5）Get。

关系：org.apache.hadoop.hbase.client.Get

作用：用来获取单个行的相关信息。

| 返回值 | 函数 | 描述 |
| --- | --- | --- |
| Get | addColumn(byte[] family, byte[] qualifier) | 获取指定列族和列修饰符对应的列 |
| Get | addFamily(byte[] family) | 通过指定的列族获取其对应列的所有列 |
| Get | setTimeRange(long minStamp,long maxStamp) | 获取指定区间的列的版本号 |
| Get | setFilter(Filter filter) | 当执行 Get 操作时设置服务器端的过滤器 |

用法示例：

```
HTable table = new HTable(conf, Bytes.toBytes(tablename));
Get g = new Get(Bytes.toBytes(row));
```

### 【案例 8-4】Hbase 的 Java API 操作之创建删除表

（1）创建 Java 工程，在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”新建一个项目“CreateDelTable”，如图 8-29 所示。

（2）创建 Java 类，在项目 src 目录下，单击右键，选择“New”创建一个类文件名称为“CreateTable”，并指定包名“com.simple.create”，如图 8-30 所示。

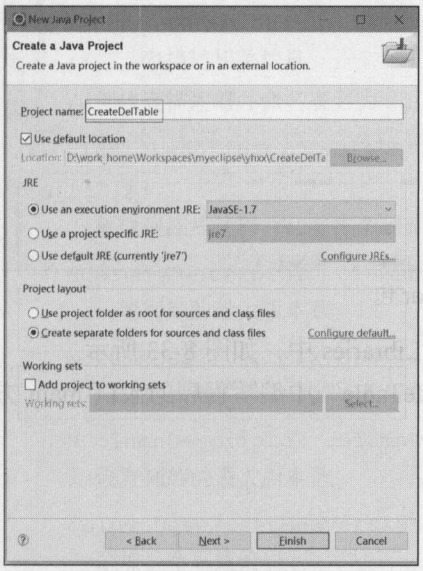



图 8-29　新建项目

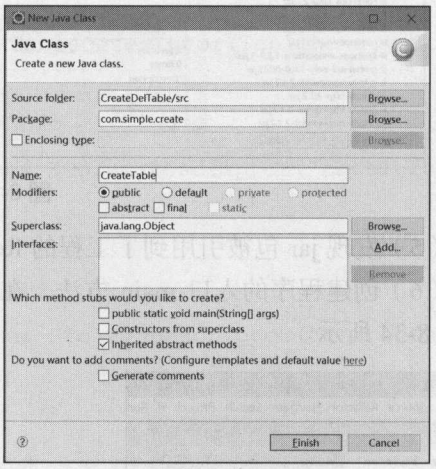



图 8-30　新建包名和类

（3）复制 Hbase 相关 jar 包到 lib 文件夹，在编写“CreateTable”类之前需要把 Hbase 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib，把 Hbase 相关 jar 包复制到该文件夹中，如图 8-31 所示。

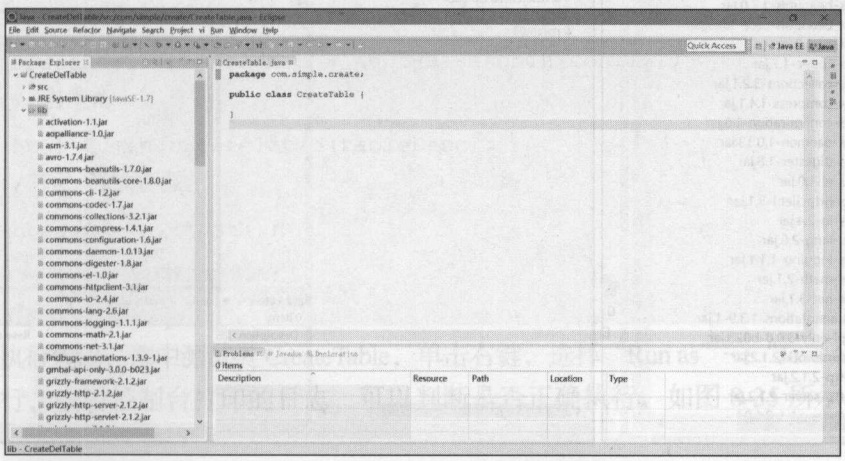



图 8-31　导入 jar 包

（4）将 lib 下所有的 jar 包导入到项目环境中，首先全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”，添加后，发现在 lib 文件夹下的 jar 包图标变成了瓶子状，如图 8-32 所示。

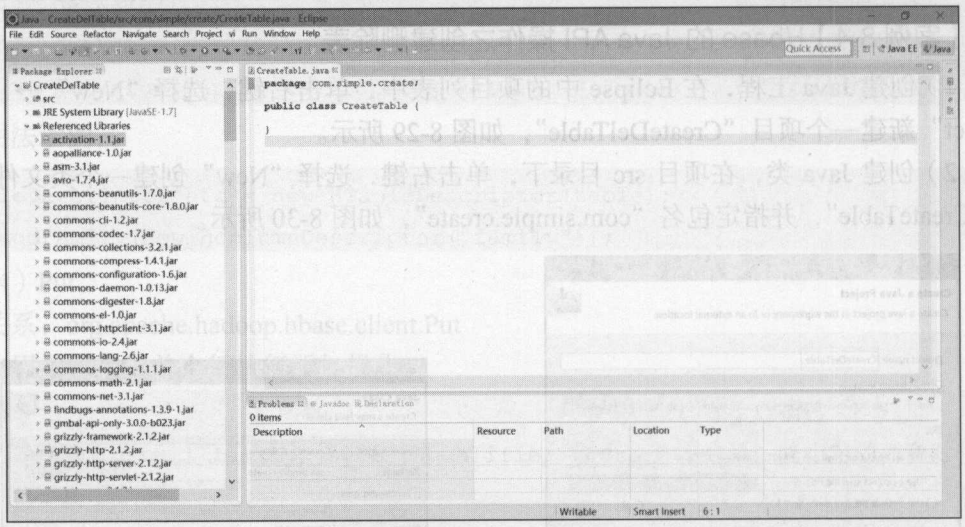



图 8-32　添加 jar 包

（5）发现 jar 包被引用到了工程的 Referenced Libraries 中，如图 8-33 所示。

（6）创建程序的入口 main 方法，在类“CreateTable”中编写程序的入口 main 方法，如图 8-34 所示。

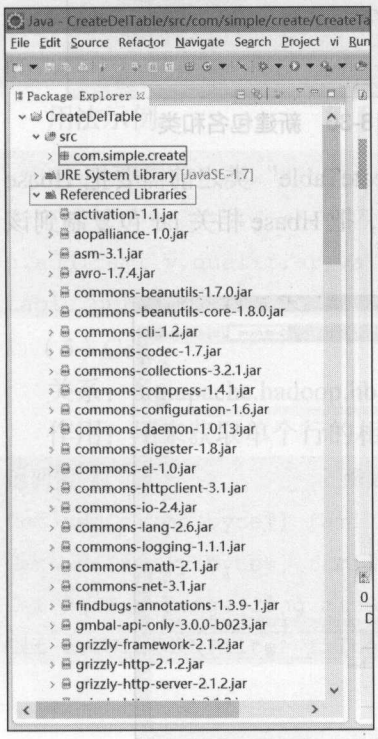



图 8-33　引用 jar 包

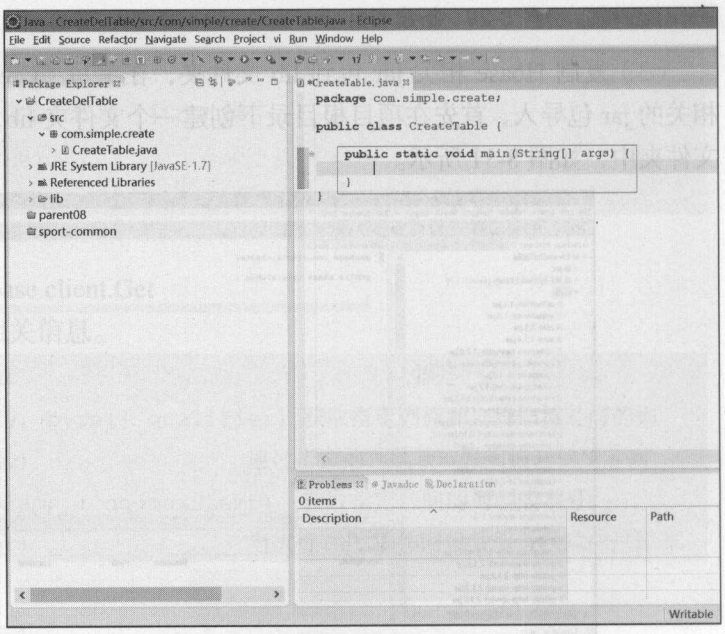



图 8-34　类

（7）代码如下所示。

```
package com.simple.create;
public class CreateTable {
        public static void main(String[] args) throws IOException {
        //一、配置文件设置
        //创建用于客户端的配置类实例
        Configuration config = HBaseConfiguration.create();
        //设置连接 zookeeper 的地址
        //hbase 客户端连接的是 zookeeper
        config.set("hbase.zookeeper.quorum", "192.168.0.131:2181");
        //二、表描述相关信息
        //创建表描述器并命名表名为 account1
        HTableDescriptor tableDesc = new HTableDescriptor(TableName.valueOf
("account1"));
        //创建列族描述器并命名一个列族名为 baseInfo
        HColumnDescriptor columnDesc1 = new HColumnDescriptor("baseinfo");
        //设置列族的最大版本数
        columnDesc1.setMaxVersions(5);
        //创建列族描述器并命名一个列族名为 baseInfo
        HColumnDescriptor columnDesc2 = new HColumnDescriptor("contacts");
        //设置列族的最大版本数
        columnDesc2.setMaxVersions(3);
        //添加一个列族给表
        tableDesc.addFamily(columnDesc1);
        //添加一个列族给表
        tableDesc.addFamily(columnDesc2);
        //三、实例化 HBaseAdmin、创建表
        //根据配置文件创建 HBaseAdmin 对象
        HBaseAdmin hbaseAdmin = new HBaseAdmin(config);
        //创建表
        hbaseAdmin.createTable(tableDesc);
        //四、释放资源
        hbaseAdmin.close();
    }
}
```

（8）执行代码，选中测试类 CreateTable，单击右键，选择“Run as”→“Java Application”，程序将执行，查看控制台打印的日志，可以判断是否正确执行，如图 8-35 所示。

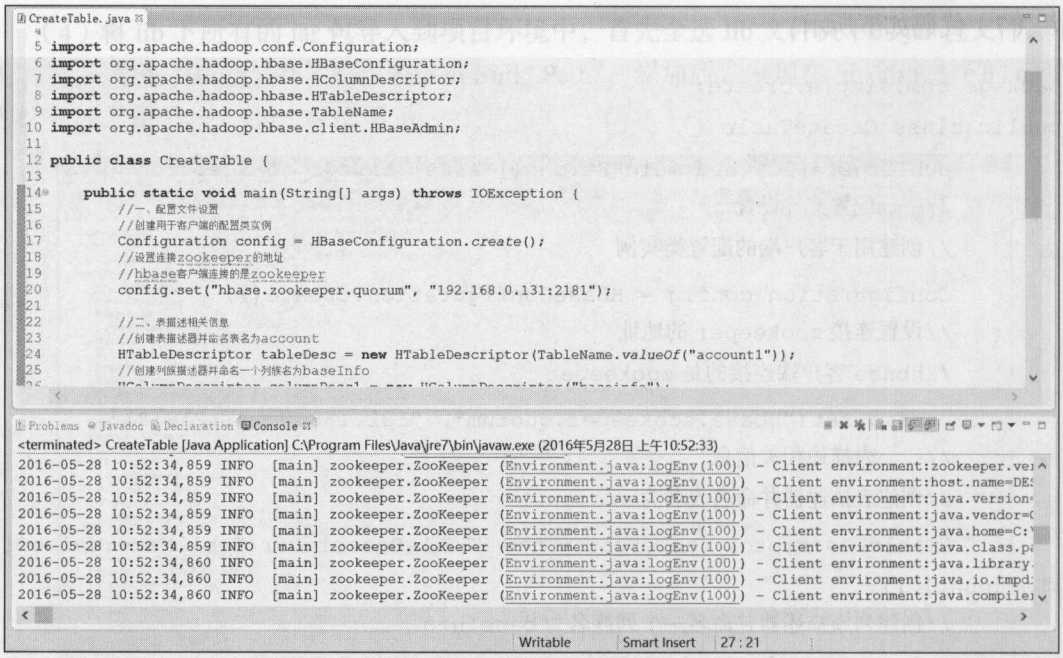



图 8-35　控制台信息

（9）如果控制台打印如下日志 java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries，无需理会。该异常是 window 平台没有查找到 winutils.exe 所打印的提示，如图 8-36 所示。

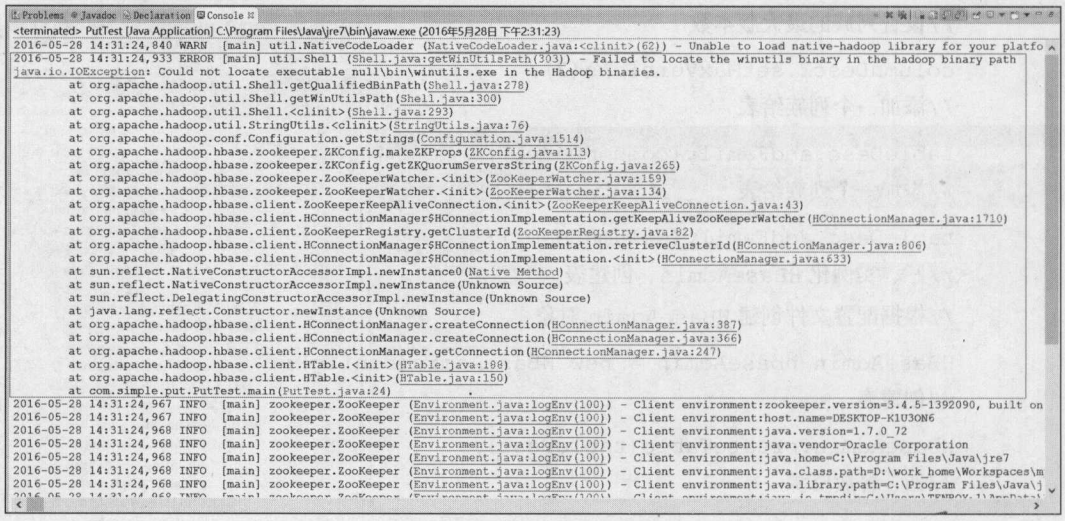



图 8-36　异常处理

（10）进入 Hbase Shell 查看结果，程序执行完毕之后，进入 Hbase Shell，分别执行命令 list 和 describe 'account1'，可以查看执行的结果，如图 8-37 所示。

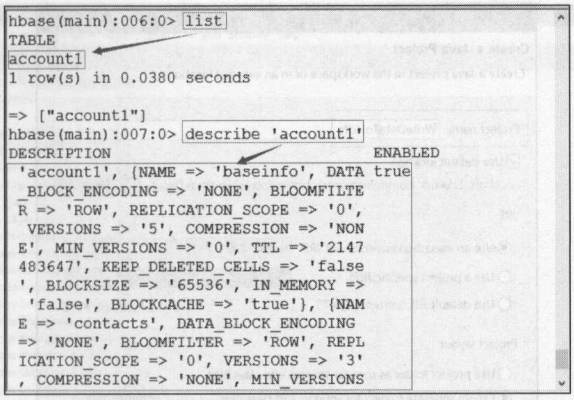



图 8-37 删除表

（11）删除表的操作步骤同上。删除表代码，如下所示。

```
package com.simple.create;
public class DelTable {
        public static void main(String[] args) throws IOException {
        //一、配置文件设置
        //创建用于客户端的配置类实例
        Configuration config = HBaseConfiguration.create();
        //设置连接 zookeeper 的地址
        //hbase 客户端连接的是 zookeeper
        config.set("hbase.zookeeper.quorum", "192.168.0.131:2181");

        //二、实例化 HBaseAdmin、创建表
        //根据配置文件创建 HBaseAdmin 对象
        HBaseAdmin hbaseAdmin = new HBaseAdmin(config);
        //三、删除表
        //设置表 account1 不可用(删除表之前需要先设置表不可用)
        hbaseAdmin.disableTable("account1");
        //删除表 account1
        hbaseAdmin.deleteTable("account1");
        //四、释放资源
        hbaseAdmin.close();
    }
}
```

【案例 8-5】从 Hbase 中读取数据写入 HDFS 中

（1）创建 Java 工程，在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”新建一个项目“WriteDataToHdfs”，如图 8-38 所示。

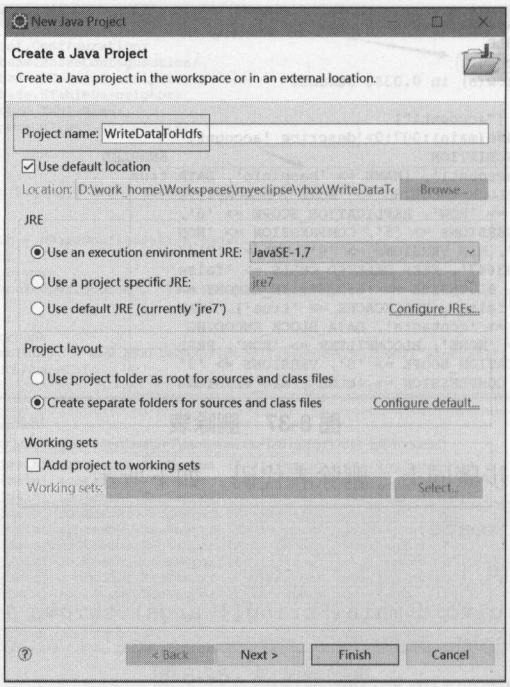



图 8-38　新建项目

（2）复制 Hbase 相关 jar 包到 lib 文件夹，首先在项目根目录下创建一个文件夹 lib，把 Hbase 相关 jar 包复制到该文件中，如图 8-39 所示。

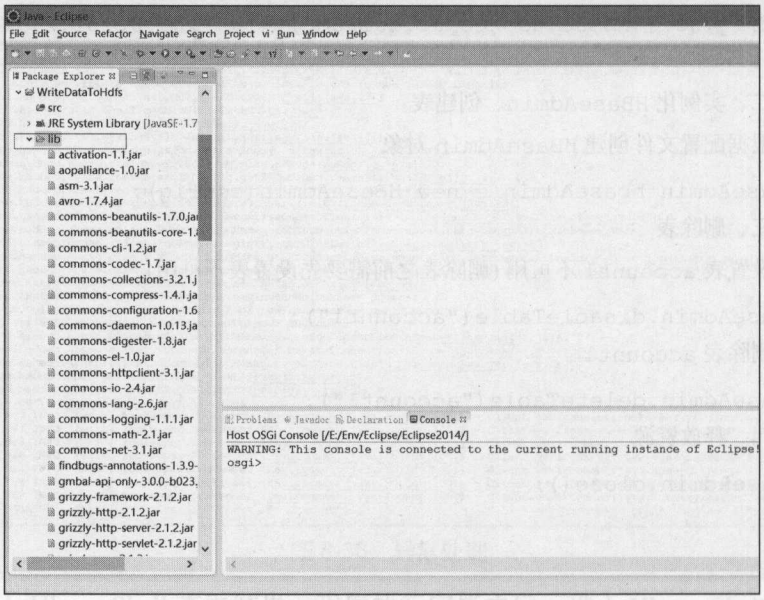



图 8-39　导入 jar 包

（3）将 lib 下所有的 jar 包导入到项目环境中，首先全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”。添加后，发现 jar 包被引用到了工程的 Referenced Libraries 中，如图 8-40 所示。

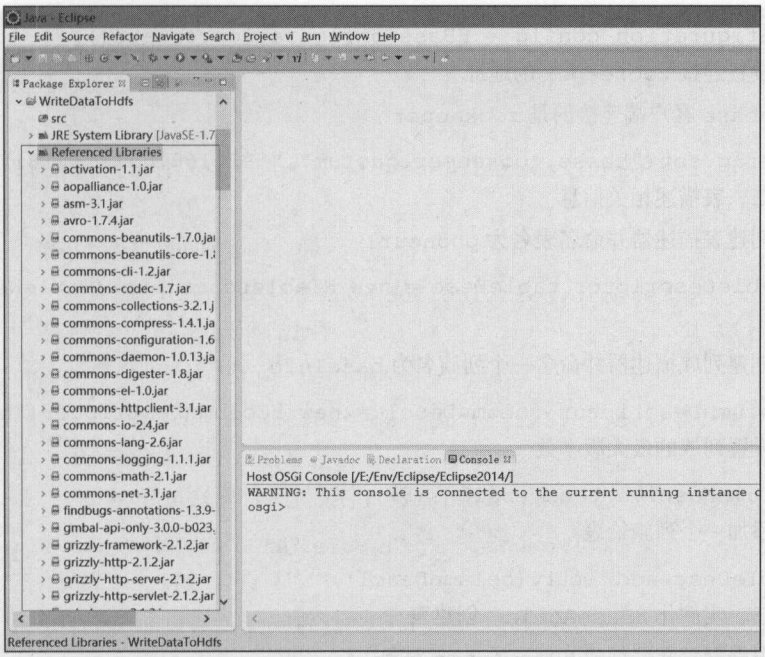



图 8-40　引用 jar 包

（4）创建一个来建表的 Java 类，在项目 src 目录下，单击右键，选择“New”创建一个类文件名称为“CreateTable”的 Java 类，并指定包名“com.simple”，如图 8-41 所示。

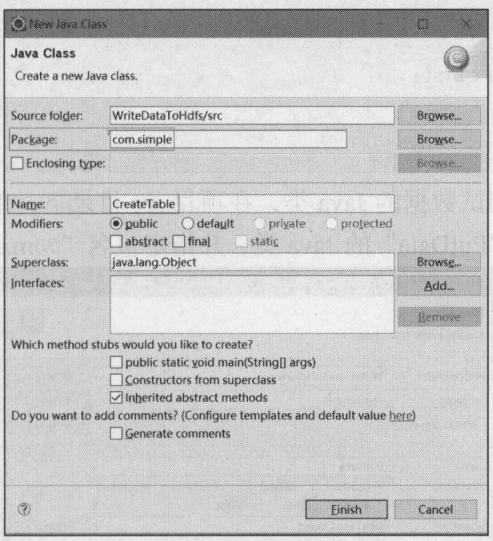



图 8-41　新建包名

（5）编写代码如下，如下所示。

```
package com.simple;
public class CreateTable {
    public static void main(String[] args) throws IOException {
        //一、配置文件设置
        //创建用于客户端的配置类实例
```

```
        Configuration config = HBaseConfiguration.create();
        //设置连接 zookeeper 的地址
        //hbase 客户端连接的是 zookeeper
        config.set("hbase.zookeeper.quorum","192.168.0.131:2181");
        //二、表描述相关信息
        //创建表描述器并命名表名为 phoneurl
        HTableDescriptor tableDesc = new HTableDescriptor(TableName.valueOf
("phoneurl"));
        //创建列族描述器并命名一个列族名为 baseInfo
        HColumnDescriptor columnDesc1 = new HColumnDescriptor("baseinfo");
        //设置列族的最大版本数
        columnDesc1.setMaxVersions(5);
        //添加一个列族给表
        tableDesc.addFamily(columnDesc1);
        //三、实例化 HBaseAdmin、创建表
        //根据配置文件创建 HBaseAdmin 对象
        HBaseAdmin hbaseAdmin = new HBaseAdmin(config);
        //创建表
        hbaseAdmin.createTable(tableDesc);
        //四、释放资源
        hbaseAdmin.close();
    }
}
```

（6）创建用于添加测试数据的 Java 类，在项目 src 目录下，单击右键，选择“New”，创建一个类文件名称为“PutData”的 Java 类，并指定包名“com.simple”，如图 8-42 所示。

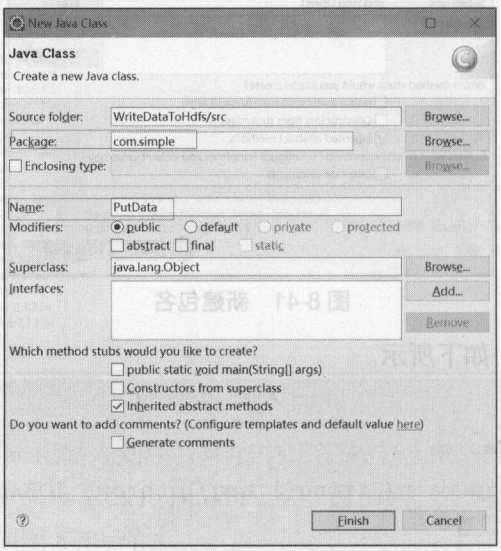



图 8-42　类名 1

（7）编写代码，如下所示。

```
package com.simple;
public class PutData {
    public static void main(String[] args) throws IOException {
        // 一、配置文件设置
        // 创建用于客户端的配置类实例
        Configuration config = HBaseConfiguration.create();
        // 设置连接 zookeeper 的地址
        // hbase 客户端连接的是 zookeeper
        config.set("hbase.zookeeper.quorum","192.168.0.131:2181");
        // 二、 获得要操作的表的对象。
        // 第一个参数"config"为配置文件；第二个参数"phoneurl"为数据库中的表名。
        HTable table = new HTable(config, "phoneurl");
        // 三、设置 Put 对象
        // 设置行键值 ；设置列族、列、cell 值
        Put put1 = new Put(Bytes.toBytes("15901235350"));
        put1.add(Bytes.toBytes("baseinfo"), Bytes.toBytes("url"),
                Bytes.toBytes("www.ifeng.com"));
        Put put2 = new Put(Bytes.toBytes("15901235351"));
        put2.add(Bytes.toBytes("baseinfo"), Bytes.toBytes("url"),
                Bytes.toBytes("www.so.com"));
        Put put3 = new Put(Bytes.toBytes("15901235352"));
        put3.add(Bytes.toBytes("baseinfo"), Bytes.toBytes("url"),
                Bytes.toBytes("www.bing.com"));
        // 四、构造 List<Put>
        List<Put> listPut = new ArrayList<Put>();
        listPut.add(put1);
        listPut.add(put2);
        listPut.add(put3);
        // 五、插入多行数据
        table.put(listPut);
        // 六、释放资源
        table.close();
    }
}
```

（8）创建用于将数据写入到 HDFS 的 Java 类，在项目 src 目录下，单击右键，选择“New”，创建一个类文件名称为“WriteDataToHdfs”的 Java 类，并指定包名“com.simple”，如图 8-43 所示。

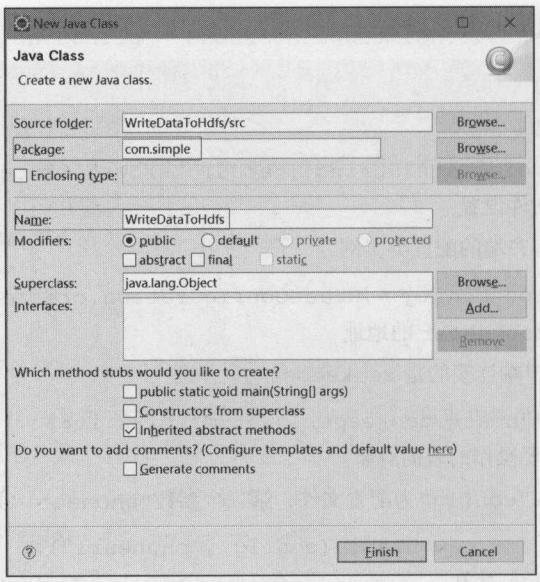



图 8-43　类名 2

（9）代码如下所示。

```
package com.simple;
public class WriteDataToHdfs {
    //hbase 表名
    public static String tableName = "phoneurl";
    //HdfsSinkMapper 继承自 TableMapper，TableMapper 继承自 Mapper 类。
    static class HdfsSinkMapper extends TableMapper<Text, NullWritable>{
        /*
         * map  <br/>
         * 拿到一行的内容。<br/>
         * 参数 key 代表 hbase 行键；参数 result 代表一行字段的值。<br/>
         */
        @Override
        protected void map(ImmutableBytesWritable key, Result value, Context
context) throws IOException, InterruptedException {
            //hbase 的 rowkey
            //获取 key 的内容
            byte[] bytes = key.copyBytes();
            //rowkey 是手机号
            //将 key 转换为 String
            String phone = new String(bytes);
            //value 为 url 字段，属于行键 baseinfo
            //根据列族名和列名获取列值
```

```
            byte[] urlbytes = value.getValue("baseinfo".getBytes(),"url".
getBytes());
            //将列值转换为String
            String url = new String(urlbytes);
            // 将一行数据写出去
            context.write(new Text(phone + "\t" + url), NullWritable.get());
        }
    }
    /**
     * reduce <br/>
     * reduce里边什么也没有做，也就是把map里边的一行数据继续写出去。
     */
    static class HdfsSinkReducer extends Reducer<Text, NullWritable, Text,
NullWritable>{
        @Override
        protected void reduce(Text key, Iterable<NullWritable> values,
Context context) throws IOException, InterruptedException {
            context.write(key, NullWritable.get());
        }
    }
    public static void main(String[] args) throws Exception {
        // 配置文件设置，创建用于客户端的配置类实例
        Configuration conf = HBaseConfiguration.create();
        // 设置连接zookeeper的地址
        conf.set("hbase.zookeeper.quorum", "192.168.0.131:2181");
        //获取Job实例
        Job job = Job.getInstance(conf);
        //为Job设置HbaseReader
        job.setJarByClass(WriteDataToHdfs.class);
        Scan scan = new Scan();
        //设置mapper
        //第一个参数为表名；第二个参数为Scan，要写表中所有的数据，所以可以指定起始的行
键范围;第三个参数为mapper类;第四个参数为输出的keyclass;第五个参数为输出的valueclass。
        //去访问表 phoneurl; Scan: 给的是全表扫描; HdfsSinkMapper.class: 上面写的
mapper类; Text.class+NullWritable.class: mapper的输出; job
        TableMapReduceUtil.initTableMapperJob(tableName, scan, HdfsSinkMapper.
class, Text.class, NullWritable.class, job);
        //设置reducer
        job.setReducerClass(HdfsSinkReducer.class);
```

```
        //设置 OutputFormat 的输出路径。
        FileOutputFormat.setOutputPath(job,  new  Path("d:/hbasetest/output
"));
        //设置 reducer 的 OutputKey Class
        job.setOutputKeyClass(Text.class);
        //设置 reducer 的 OutputValue Class
        job.setOutputValueClass(NullWritable.class);
        job.waitForCompletion(true);
    }
}
```

（10）执行代码，分别选择类 CreateTable、PutData、WriteDataToHdfs，先后单击右键选择“Run as”→“Java Application”，程序将执行，会完成对表的建立、向表中插入数据、向 HDFS 中写入表中的数据，如图 8-44 所示。

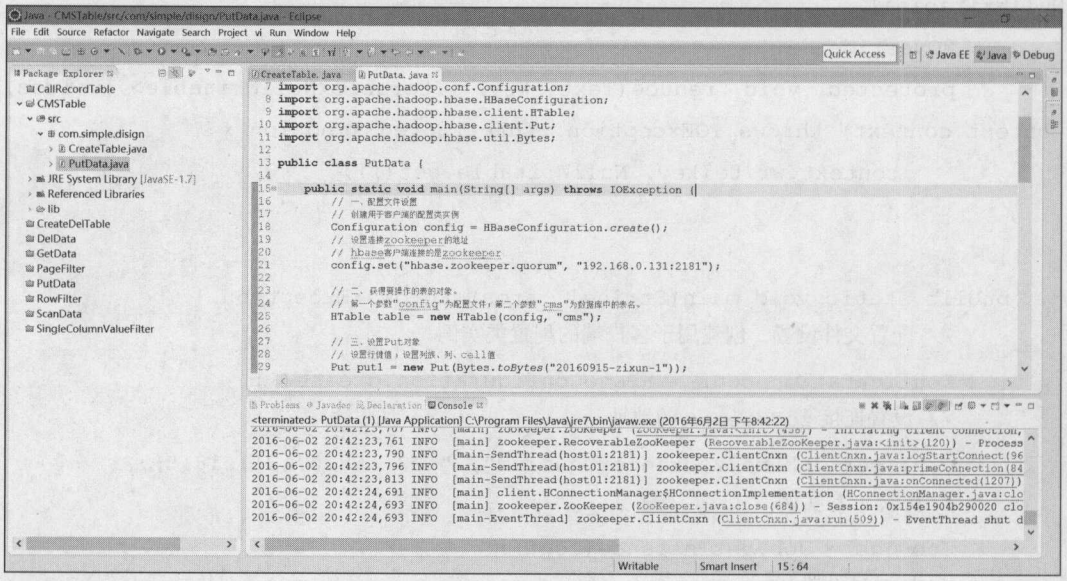



图 8-44　控制台信息

## 8.5　Hbase 过滤器

### 8.5.1　过滤器的含义

Hbase 为筛选数据提供了一组过滤器，通过这个过滤器可以在 HBase 中的数据的多个维度（行、列、数据版本）上进行对数据的筛选操作，也就是说过滤器最终能够筛选的数据能够细化到具体的一个存储单元格上（由行键、列名、时间戳定位）。通常来说，通过行键、值来筛选数据的应用场景较多。

### 8.5.2　过滤器的比较操作符

要完成过滤操作，需要两个操作，一个是抽象的操作符，Hbase 提供了枚举类型的变

量来表示这些抽象的操作符，如 LESS/LESS_OR_EQUAL/EQUAL/NOT_EUQAL 等，如表 8-1 所示；另外一个就是具体的比较器（Comparator），代表具体的比较逻辑，可以提高字节级的比较、字符串级的比较等。有了这两个参数，我们就可以清晰地定义筛选的条件，过滤数据。

表 8-1 比较操作符

| 运 算 符 | 描 述 |
| --- | --- |
| LESS | 小于 |
| LESS_OR_EQUAL | 小于等于 |
| EQUAL | 等于 |
| NOT_EQUAL | 不等于 |
| GREATER_OR_EQUAL | 大于等于 |
| GREATER | 大于 |
| NO_OP | 排除所有 |

### 8.5.3 过滤器的比较器

CompareFilter 是高层的抽象类，下面我们将看到它的实现类和实现类代表的各种过滤条件，比较器主要有 6 种，分别是 BinaryComparator、BinaryPrefixComparator、NullComparator、BitComparator、RegexStringComparator、SubstringComparator，具体如表 8-2 所示。

表 8-2 比较器

| 运 算 符 | 描 述 |
| --- | --- |
| BinaryComparator | 按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[]) |
| BinaryPrefixComparator | 和 BinaryComparator 差不多，从前面开始比较 |
| NullComparator | 判断给定的是否为空 |
| BitComparator | 按位比较 a BitwiseOp class 做异或、与、并操作 |
| RegexStringComparator | 提供一个正则的比较器，仅支持 EQUAL 和非 EQUAL |
| SubstringComparator | 判断提供的子串是否出现在 table 的 value 中 |

**【案例 8-6】Hbase 的比较过滤器 RowFilter 的使用**

（1）创建 Java 工程，在 Eclipse 中的项目列表中，单击右键，选择“New”→“Java Project”新建一个项目“RowFilter”，如图 8-45 所示。

（2）创建 Java 类，在项目 src 目录下，单击右键，选择“New”，创建一个类文件名称为“RowFilterTest”的 Java 类，并指定包名“com.simple.filter”，如图 8-46 所示。

（3）复制 Hbase 相关 jar 包到 lib 文件夹，在编写“RowFilterTest”类之前需要把 Hbase 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib，把 Hbase 相关 jar 包复制到该

文件夹中，如图 8-47 所示。

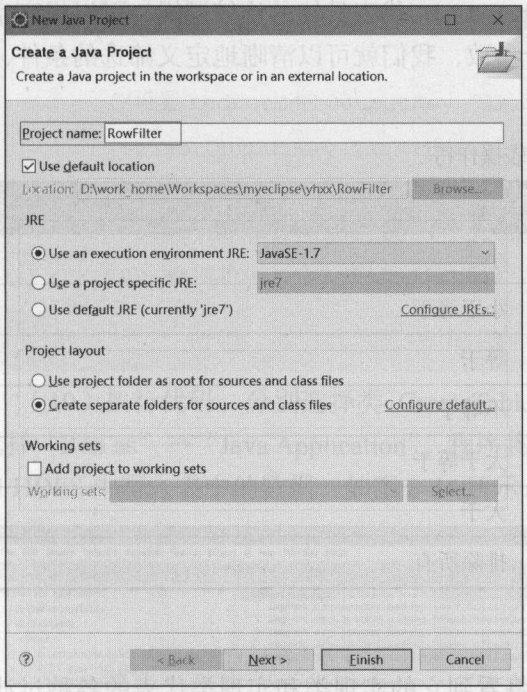



图 8-45　新建项目

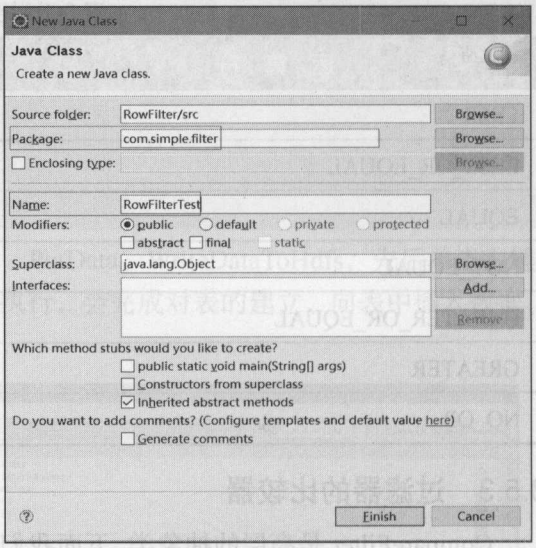



图 8-46　新建包名

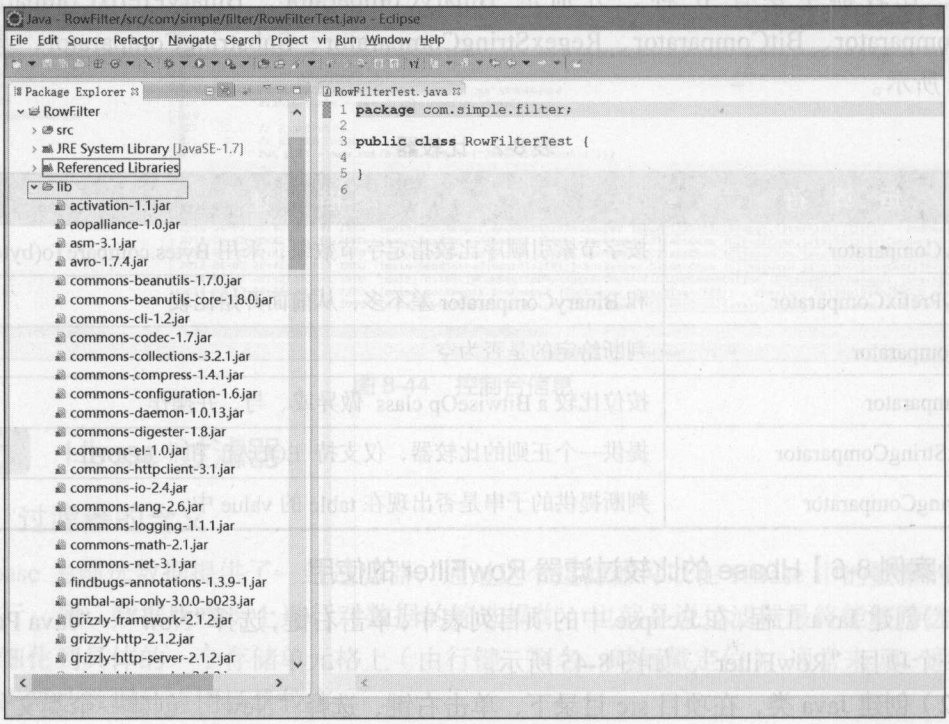



图 8-47　导入 jar 包

（4）将 lib 下所有的 jar 包导入到项目环境中，首先全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”。添加后，发现 jar 包被引用到了工程的

Referenced Libraries 中，如图 8-48 所示。

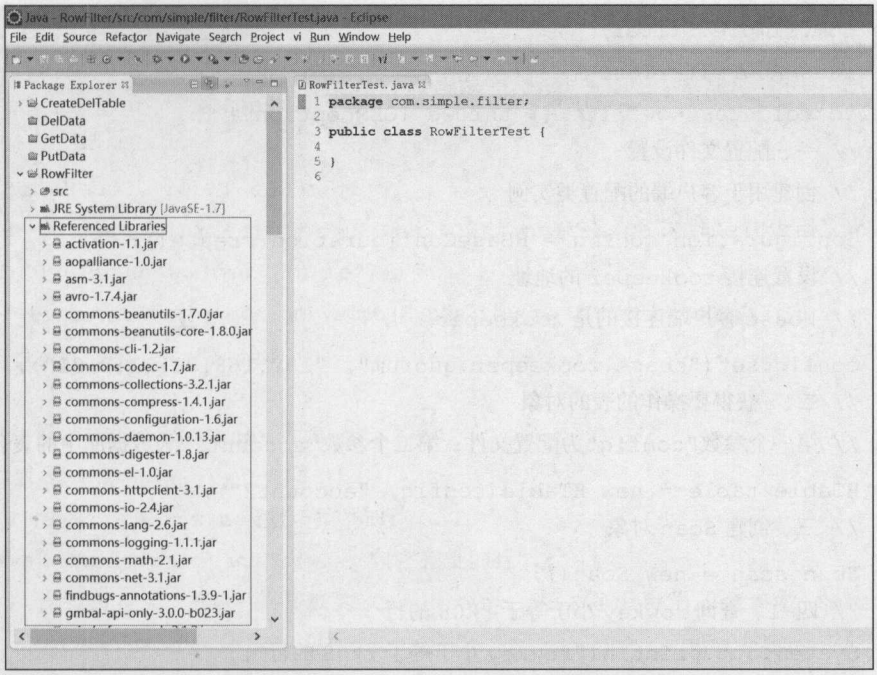



图 8-48　引入 jar 包

（5）创建程序的入口 main 方法，在类“RowFilterTest”中编写程序的入口 main 方法，如图 8-49 所示。

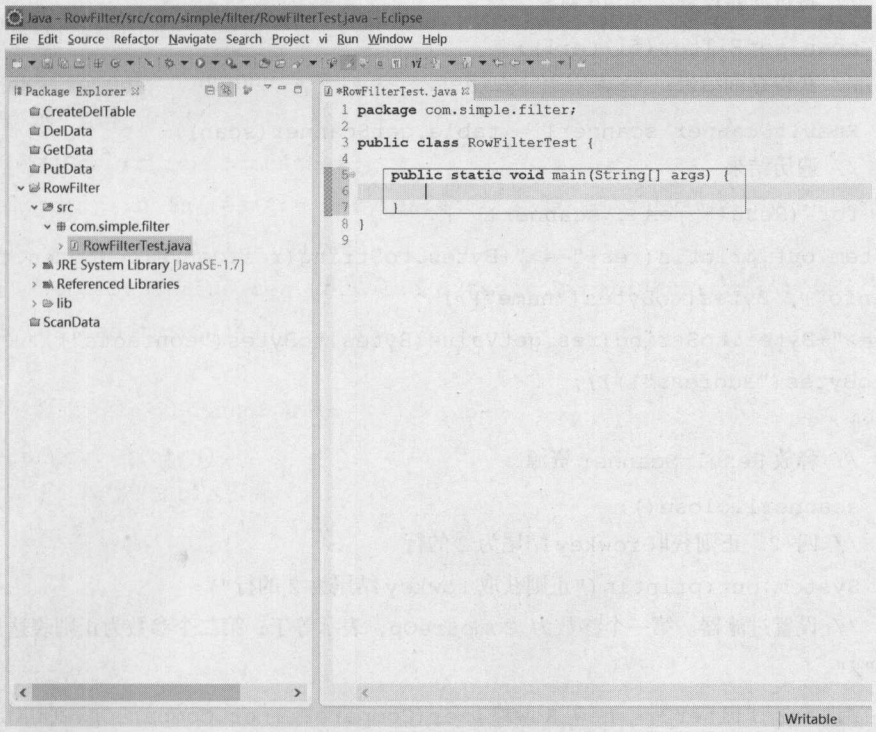



图 8-49　类

（6）编写代码如下。

```
package com.simple.filter;
public class RowFilterTest {
    public void testRowFilter() throws IOException {
        // 一、配置文件设置
        // 创建用于客户端的配置类实例
        Configuration config = HBaseConfiguration.create();
        // 设置连接 zookeeper 的地址
        // Hbase 客户端连接的是 zookeeper
        config.set("hbase.zookeeper.quorum", "192.168.0.131:2181");
        // 二、 获得要操作的表的对象
        // 第一个参数"config"为配置文件；第二个参数"account2"为数据库中的表名。
        HTable table = new HTable(config, "account2");
        // 三、创建 Scan 对象
        Scan scan = new Scan();
        // 四-1、查询 rowkey 小于等于 rk08 的行
        System.out.println("rowkey 小于等于 rk08 的行");
    // 设置过滤器。第一个参数为 CompareOp，表示小于或等于；第二个参数为行键值。
        Filter filter1 = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,
                new BinaryComparator("rk08".getBytes()));
        // 将过滤器设置给 Scan 对象
        scan.setFilter(filter1);
        // 获得查询结果
        ResultScanner scanner1 = table.getScanner(scan);
        // 遍历结果
        for (Result res : scanner1) {
    System.out.println(res+"==>"+Bytes.toString(res.getValue(Bytes.toBytes
("baseinfo"), Bytes.toBytes("name")))
    +"==>"+Bytes.toString(res.getValue(Bytes.toBytes("contacts"),
Bytes.toBytes("address"))));
        }
        // 释放 ResultScanner 资源
        scanner1.close();
        // 四-2、正则获取 rowkey 结尾为 2 的行
        System.out.println("正则获取 rowkey 结尾为 2 的行");
        // 设置过滤器。第一个参数为 CompareOp，表示等于；第二个参数为正则表达式，表示
结尾含有"2"。
        Filter filter2 = new RowFilter(CompareFilter.CompareOp.EQUAL,
                new RegexStringComparator(".*2$"));
```

```
        // 将过滤器设置给 Scan 对象
        scan.setFilter(filter2);
        // 获得查询结果
        ResultScanner scanner2 = table.getScanner(scan);
        // 遍历结果
        for (Result res : scanner2) {
    System.out.println(res+"==>"+Bytes.toString(res.getValue(Bytes.toBytes
("baseinfo"), Bytes.toBytes("name")))
    +"==>"+Bytes.toString(res.getValue(Bytes.toBytes("contacts"),
Bytes.toBytes("address"))));
        }
        // 释放 ResultScanner 资源
        scanner2.close();
        // 四-3、获取 rowkey 包含有 1 的行
        System.out.println("rowkey 包含有 1 的行");
        // 设置过滤器。第一个参数为 CompareOp，表示等于；第二个参数 Substring
Comparator，表示字符串中含有“1”。
        Filter filter3 = new RowFilter(CompareFilter.CompareOp.EQUAL,
                new SubstringComparator("1"));
        // 将过滤器设置给 Scan 对象
        scan.setFilter(filter3);
        // 获得查询结果
        ResultScanner scanner3 = table.getScanner(scan);
        // 遍历结果
        for (Result res : scanner3) {
    System.out.println(res+"==>"+Bytes.toString(res.getValue(Bytes.toBytes
("baseinfo"), Bytes.toBytes("name")))
    +"==>"+Bytes.toString(res.getValue(Bytes.toBytes("contacts"),
Bytes.toBytes("address"))));
        }
        // 释放 ResultScanner 资源
        scanner3.close();
        // 五、释放 HTable 资源
        table.close();
    }
    /**
     * @param args
     * @throws IOException
     */
```

```
    public static void main(String[] args) throws IOException {
        // 创建测试类实例
        RowFilterTest test = new RowFilterTest();
        // 调用测试代码
        test.testRowFilter();
    }
}
```

（7）执行代码，选中测试类 RowFilterTest，单击右键选择“Run as”→“Java Application”，程序将执行。查看控制台打印的日志，可以查看到运行结果，如图 8-50 所示。

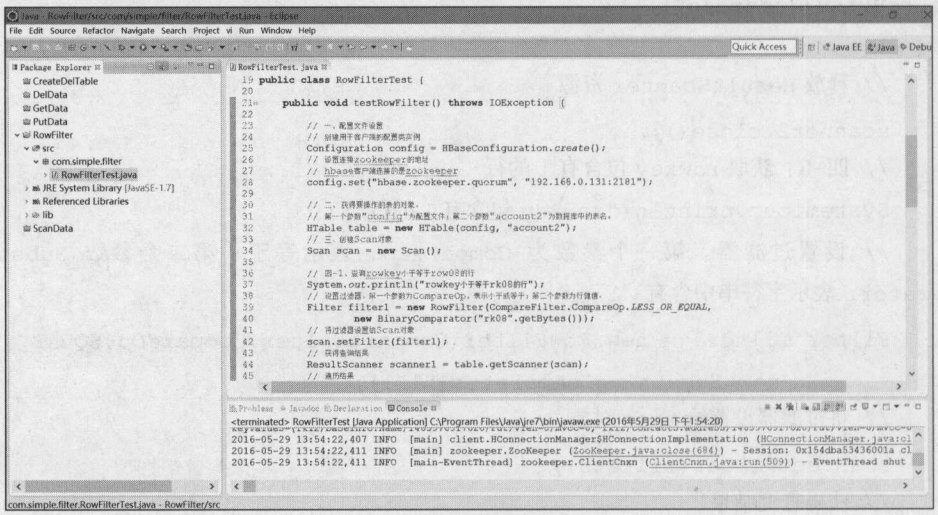


图 8-50　控制台信息

（8）如果控制台打印如下日志 java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries，无需理会。该异常是 window 平台没有查找到 winutils.exe 所打印的提示，如图 8-51 所示。

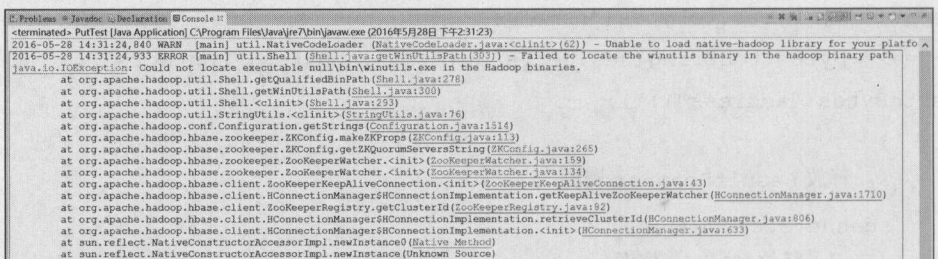


图 8-51　异常信息

（9）查看结果，控制台打印结果如图 8-52 所示。

```
rowkey小于等于rk08的行
keyvalues={rk01/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk01/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang1==>北京
keyvalues={rk02/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk02/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang2==>上海
keyvalues={rk03/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk03/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang3==>广州
keyvalues={rk04/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk04/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang4==>深圳
keyvalues={rk05/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk05/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang5==>天津
keyvalues={rk06/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk06/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang6==>郑州
keyvalues={rk07/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk07/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang7==>兰州
keyvalues={rk08/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk08/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang8==>拉萨
正则获取rowkey结尾为2的行
keyvalues={rk02/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk02/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang2==>上海
keyvalues={rk12/baseinfo:name/1463976917620/Put/vlen=8/mvcc=0, rk12/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang12==>西宁
rowkey包含有1的行
keyvalues={rk01/baseinfo:name/1463976917620/Put/vlen=7/mvcc=0, rk01/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang1==>北京
keyvalues={rk10/baseinfo:name/1463976917620/Put/vlen=8/mvcc=0, rk10/contacts:address/1463976917620/Put/vlen=12/mvcc=0}==>JiKang10==>呼和浩特
keyvalues={rk11/baseinfo:name/1463976917620/Put/vlen=8/mvcc=0, rk11/contacts:address/1463976917620/Put/vlen=12/mvcc=0}==>JiKang11==>鄂尔多斯
keyvalues={rk12/baseinfo:name/1463976917620/Put/vlen=8/mvcc=0, rk12/contacts:address/1463976917620/Put/vlen=6/mvcc=0}==>JiKang12==>西宁
```

图 8-52　控制台信息

## 【案例 8-7】Hbase 专用过滤器 PageFilter 的使用

（1）创建 Java 工程，在 Eclipse 中的项目列表中，单击右键，选择“New”→ “Java Project”新建一个项目“PageFilter”，如图 8-53 所示。

（2）创建 Java 类，在项目 src 目录下，单击右键，选择“新建”创建一个类文件名称为“PageFilterTest”的 Java 类，并指定包名“com.simple.filter”，如图 8-54 所示。

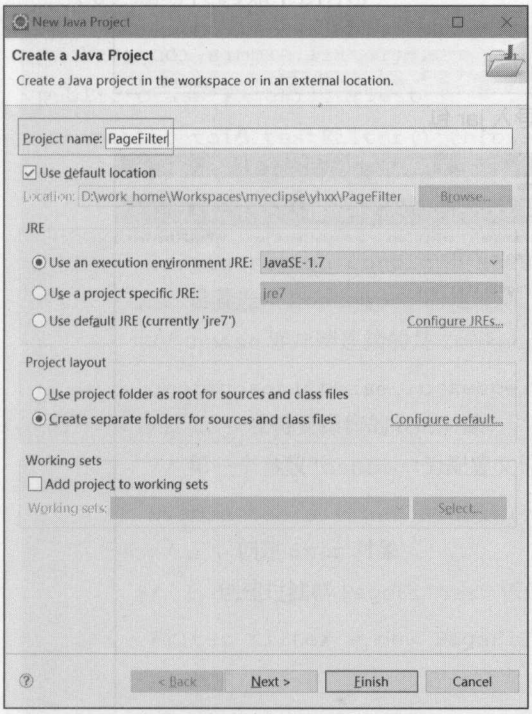



图 8-53　新建项目

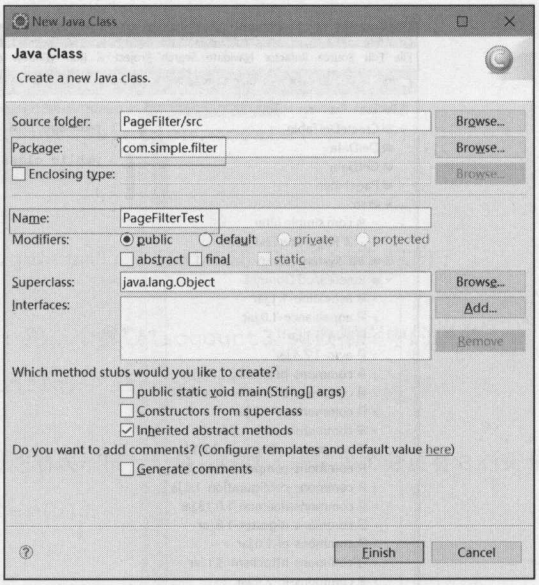



图 8-54　新建包名

（3）复制 Hbase 相关 jar 包到 lib 文件夹，在编写“PageFilterTest”类之前需要把 Hbase 相关的 jar 包导入，首先在项目根目录下创建一个文件夹 lib，把 Hbase 相关 jar 包复制到该文件夹中，如图 8-55 所示。

（4）将 lib 下所有的 jar 包导入到项目环境中，首先全选 lib 文件夹下的 jar 包文件，单击右键，选择“Build Path”→“Add to Build Path”。添加后，发现 jar 包被引用到了工程的 Referenced Libraries 中，如图 8-56 所示。

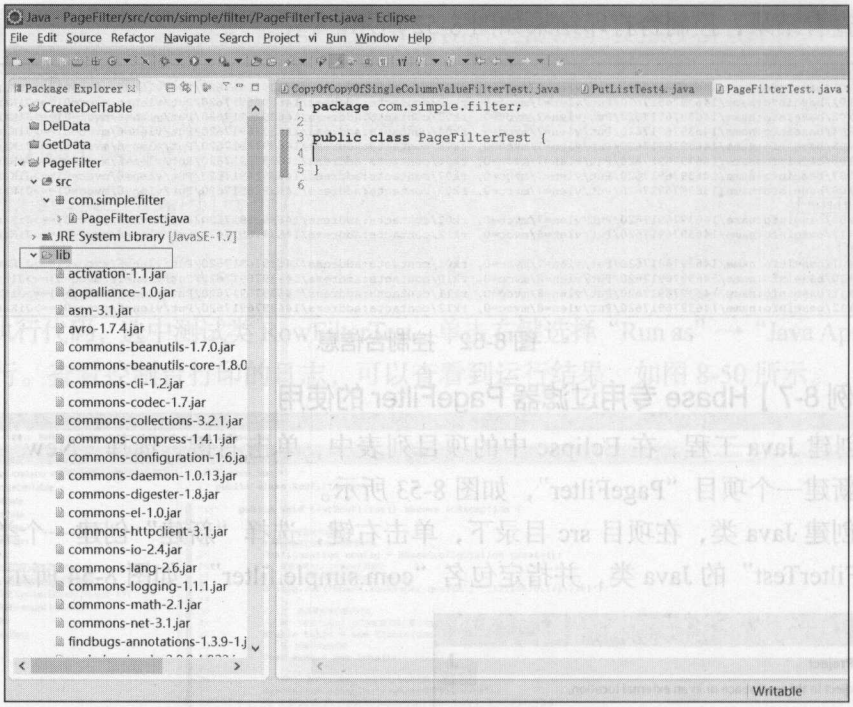



图 8-55　导入 jar 包

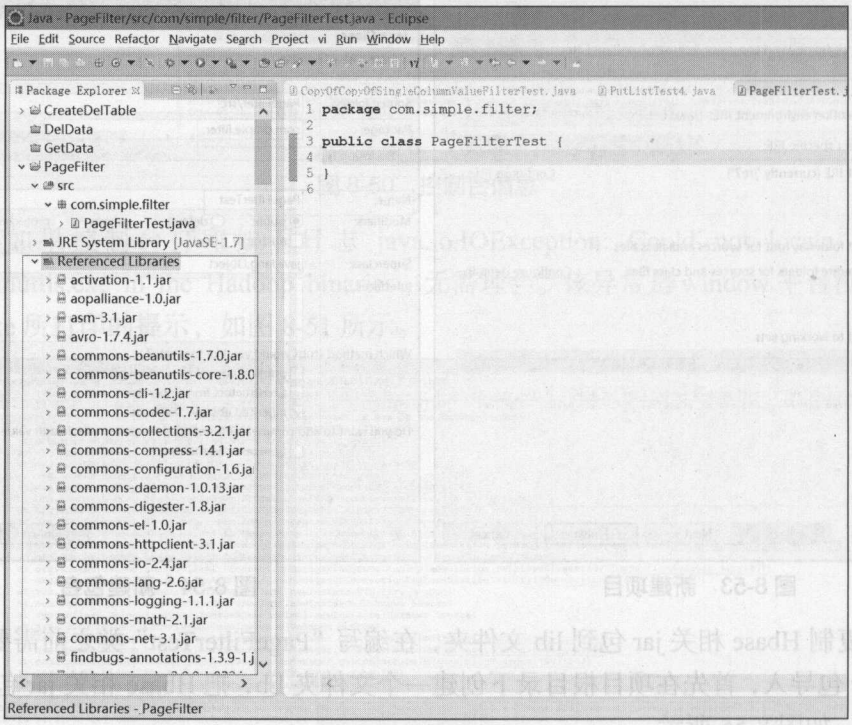



图 8-56　添加 jar 包

（5）创建程序的入口 main 方法，在类“PageFilterTest”中编写程序的入口 main 方法，如图 8-57 所示。

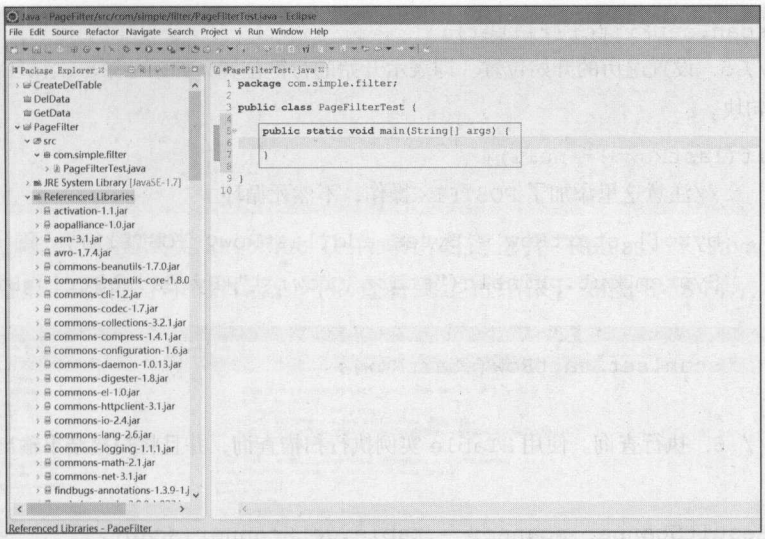



图 8-57 引用 jar 包

（6）编写代码如下所示。

```
package com.simple.filter;
public class PageFilterTest {
    public void testFilter() throws IOException {
        // 一、配置文件设置
        // 创建用于客户端的配置类实例
        Configuration config = HBaseConfiguration.create();
        // 设置连接 Zookeeper 的地址
        // hbase 客户端连接的是 Zookeeper
        config.set("hbase.zookeeper.quorum", "192.168.0.131:2181");
        // 二、 获得要操作的表的对象
        // 第一个参数"config"为配置文件；第二个参数"account3"为数据库中的表名。
        HTable table = new HTable(config, "account3");
        // 三、创建 Scan 对象
        //1. 创建过滤器 PageFilter。该过滤器表示按行分页。参数 3 表示每个分页有 3 行记录。
        Filter filter = new PageFilter(3);
        // POSTFIX=0
        final byte[] POSTFIX = new byte[] { 0x00 };
        int totalRows = 0;
        byte[] lastRow = null;
      // 2. 进入循环。为了演示效果，这里遍历所有符合条件的数据，需要循环输出。
      while (true) {
      // 3. 初始化 Scan 实例。该实例用于查询符合条件的数据。
          Scan scan = new Scan();
          // 4. 设置过滤器。将前面创建好的分页过滤器设置到 Scan 实例中。
```

```
            scan.setFilter(filter);
            // 5. 设置遍历的开始位置。即表示开始的行键位置，如果是第一次循环（即第一页），
则不进入该语句块。
            if(lastRow != null){
                //注意这里添加了 POSTFIX 操作，不然死循环了
                byte[] startRow = Bytes.add(lastRow, POSTFIX);
                System.out.println("start  row:  "+Bytes.toStringBinary(start
Row));//
                scan.setStartRow(startRow);
            }
            // 6. 执行查询。使用 HTable 实例执行扫描查询，并且将扫描结果输出，并且给行键
遍历赋值。
            ResultScanner scanner = table.getScanner(scan);
            int localRows = 0;
            Result result;
            // 输出一页的结果。
            while((result = scanner.next()) != null){
                //System.out.println(localRows++ + ":" + result);
    System.out.println(result+"==>"+Bytes.toString(result.getValue(Bytes.t
oBytes("baseinfo"), Bytes.toBytes("name")))

    +"==>"+Bytes.toString(result.getValue(Bytes.toBytes("contacts"),
Bytes.toBytes("address"))));
                totalRows ++;
                localRows ++;//
                lastRow = result.getRow();
            }
            System.out.println("");
            // 7. 关闭 ResultScanner 实例。
            scanner.close();
            // 8. 跳出循环条件
            if(localRows == 0) break;
        }
        System.out.println("total rows: " + totalRows);
         // 五、释放 HTable 资源
         table.close();
    }
    public static void main(String[] args) throws IOException {
        // 创建测试类实例
```

```
        PageFilterTest test = new PageFilterTest();
        // 调用测试代码
        test.testFilter();
    }
}
```

（7）执行代码，选中测试类 PageFilterTest，单击右键选择“Run as”→“Java Application”，程序将执行。查看控制台打印的日志，可以查看到运行结果，如图 8-58 所示。

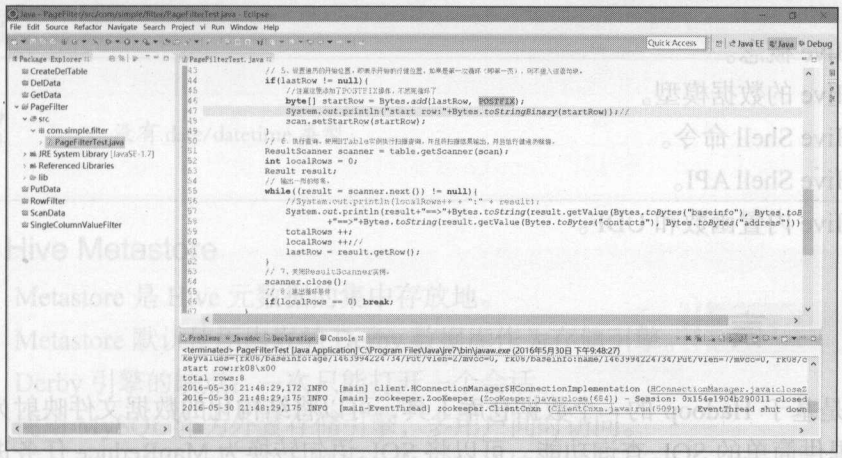


图 8-58 运行程序

（8）查看结果，控制台打印结果如图 8-59 所示。

```
keyvalues={rk01/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk01/baseinfo:name/146399422
keyvalues={rk02/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk02/baseinfo:name/146399422
keyvalues={rk03/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk03/baseinfo:name/146399422

start row:rk03\x00
keyvalues={rk04/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk04/baseinfo:name/146399422
keyvalues={rk05/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk05/baseinfo:name/146399422
keyvalues={rk06/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk06/baseinfo:name/146399422

start row:rk06\x00
keyvalues={rk07/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk07/baseinfo:name/146399422
keyvalues={rk08/baseinfo:age/1463994224734/Put/vlen=2/mvcc=0, rk08/baseinfo:name/146399422

start row:rk08\x00

total rows:8
```

图 8-59 控制台信息

## 本章小结

在本章中，首先介绍了 Hbase 的概念，然后介绍了 Hbase 的架构，并分析架构中的各个角色和所起到的作用，最后介绍 Hbase Shell 和 Hbase API 及 Hbase 过滤器的使用，让学习者通过命令和 API 接口与 Hbase 进行交互，解决数据存储问题。

## 习题

1. 简述 Hbase 数据库。
2. 简述 Hbase 架构角色。
3. 理解 Hbase 过滤器的作用。

扫一扫在线测

# 第 9 章 Hadoop 数据仓库 Hive

## 本章要点

- Hive 概念。
- Hive 的数据模型。
- Hive Shell 命令。
- Hive Shell API。
- Hive 内置函数和 UDF。

## 引　言

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行，Hive 在 Hadoop 之上提供了数据查询的能力，主要解决非关系型数据查询问题。本章通过对 Hive 的概述、Hive 的架构、Hive Shell 命令、Hive API 操作的讲解，让学生深刻理解并学会运用 Hive 系统。

## 9.1 Hive 概述

### 9.1.1 Hive 简介

Hive 本身是建立在 Hadoop 体系结构上的数据仓库基础构架，可以将结构化的数据文件映射为一张数据库表，并提供完整的 QL 语句，把 QL 语句转化成 MapReduce 程序提交给 Hadoop 集群完成相关任务。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析并存储在 Hadoop 中的大规模数据处理的机制。Hive 定义了简单的类 SQL 查询语言，称为 QL，熟悉 SQL 的用户都可以进行查询数据。同时，这个语言也允许熟悉 MapReduce 的开发者开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 以满足无法完成的复杂的分析工作。

### 9.1.2 Hive 数据类型

Hive 具有支持的数据类型，主要分为两种：基本数据类型和复杂数据类型，复杂数据类型也可以称为集合数据类型。

（1）基本数据类型，主要分为 4 类，如下所示。

```
tinyint /smallint/int/bigint
float/double
boolean
string
```

（2）复杂数据类型，主要分为 3 类，如下所示。

```
Array
Map
Struct
```

注意

没有 date/datetime 类型。

### 9.1.3 Hive Metastore

- Metastore 是 Hive 元数据的集中存放地。
- Metastore 默认使用内嵌的 Derby 数据库作为存储引擎。
- Derby 引擎的缺点：一次只能打开一个会话。
- 使用 MySQL 作为外置存储引擎，多用户同时访问。

Metastore 用来存储 Hive 的元数据信息，默认情况下是和 Hive 绑定的，部署在同一个 JVM 中，将元数据存储到 Derby 中。使用内嵌数据库没有办法为 Hive 开启多个实例。如果 Hive 配置成 MySQL 数据库，可以将数据独立出来在多个实例之间共享。

学习小贴士

Metastore 是 Hive 元数据的集中存放地。Metastore 默认使用内嵌的 Derby 数据库作为存储引擎。Derby 引擎的缺点：一次只能打开一个会话，使用 MySQL 作为外置存储引擎，多用户同时访问。

### 9.1.4 Hive 存储和压缩

首先，Hive 没有专门的数据存储格式，也没有为数据建立索引，用户可以非常自由地组织 Hive 中的表，只需要在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。

其次，Hive 中所有的数据都存储在 HDFS 中，Hive 中包含以下数据模型：Table、External Table、Partition、Bucket。

### 9.1.5 Hive 与传统数据库对比

Hive 作为大数据环境下的数据仓库工具，和传统的数据库虽然有很多相似之处，但是还是有很多不同之处，在传统的数据库中，表的模式是在数据加载的时候强行确定好的，而 Hive 在加载的过程中不对数据进行任何验证操作，加载过程比传统数据库快一些，具体其他不同之处，可以参考表 9-1。

表 9-1　Hive 与传统数据库对比

| 查询语言 | HiveQL | SQL |
| --- | --- | --- |
| 数据存储位置 | HDFS | Raw Device or 本地 FS |
| 数据格式 | 用户定义 | 系统决定 |
| 数据更新 | 不支持 | 支持 |
| 索引 | 新版本有，但弱 | 有 |
| 执行 | MapReducer | Executor |
| 执行延迟 | 高 | 低 |
| 可扩展性 | 高 | 低 |
| 数据规模 | 大 | 小 |

## 9.2 Hive 的系统架构

Hive 的系统架构组成主要分四个部分：用户接口部分，存放元数据的数据库，解释器、编译器，存放数据的 HDFS 系统，如图 9-1 所示。

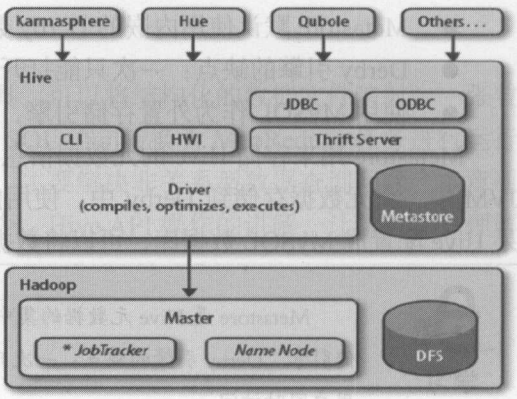



图 9-1　Hive 系统架构

- 用户接口，包括 CLI、JDBC/ODBC、WebUI。
- 元数据存储，通常是存储在关系数据库如 MySQL、DERBY 中。
- 解释器、编译器、优化器、执行器。
- Hadoop：用 HDFS 进行存储，利用 MapReduce 进行计算。

用户接口主要有三个：CLI、JDBC/ODBC 和 WebUI。

- Hive 将元数据存储在数据库中(Metastore)，目前只支持 MYSQL、DERBY。Hive 中的元数据包括表的名字、表的列和分区及其属性、表的属性（是否为外部表等）、表的数据所在目录等。
- 解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划（plan）的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行。
- Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成（包含 * 的查询，比如 select * from table 不会生成 MapReduce 任务）。

【案例 9-1】Hive 环境搭建（MySQL 数据库）

（1）在 Linux 系统下，首先执行命令 cd /simple，进入 simple 目录下，然后把/simple/soft 目录下的 Hive 压缩包解压到/simple 目录下，执行命令为 tar　-zxvf　/simple/soft/hive-0.12.0.

tar.gz，解压 Hive 压缩包，如图 9-2 所示。

（2）执行完解压命令之后，通过执行命令 ls，可以看到一个 Hive 解压目录，如图 9-3 所示。

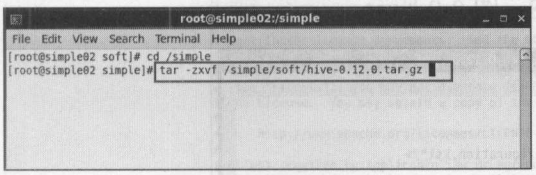



图 9-2　解压软件

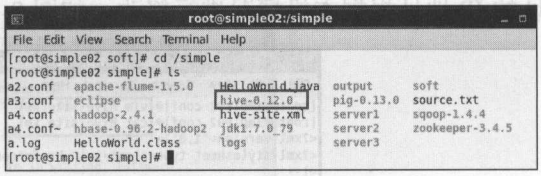



图 9-3　查看解压后软件

（3）解压完毕 Hive 压缩包后，切换目录到/simple/hive-0.12.0 并查看下面的文件列表，如图 9-4 所示。

（4）在/simple/hive-0.12.0 目录下执行命令 cd conf，切换到 conf 目录并查看列表，执行命令 cp hive-env.sh.template hive-env.sh，如图 9-5 所示。

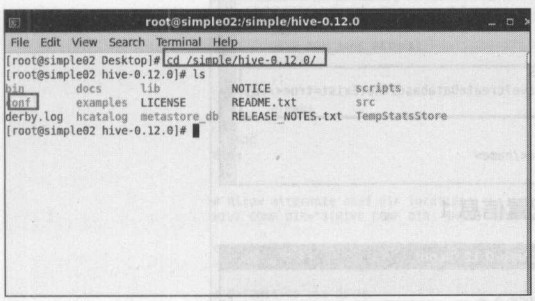



图 9-4　切换目录

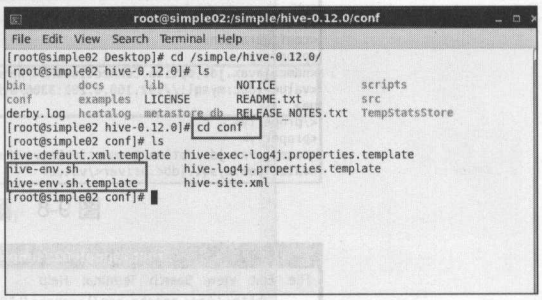



图 9-5　查看目录列表

（5）在/simple/hive-0.12.0/conf 目录下执行命令 vim hive-env.sh，进入编辑状态并编辑内容，如图 9-6 所示。

（6）在/simple/hive-0.12.0 目录下执行命令 cd conf，切换到 conf 目录并查看列表，执行命令 mv hive-default.xml.template　hive-site.xml，如图 9-7 所示。

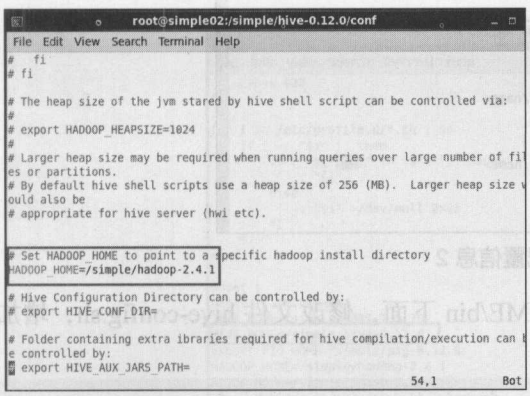



图 9-6　配置环境

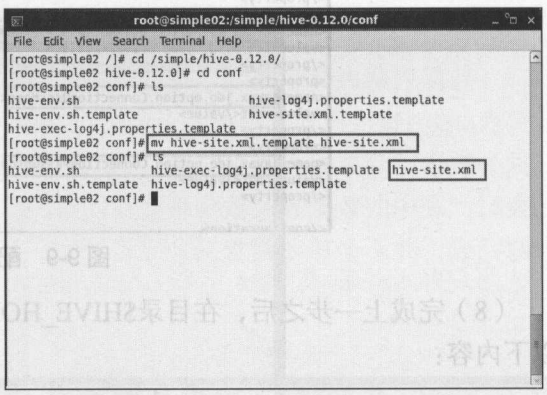



图 9-7　修改文件名

（7）完成上一步操作之后，此时需要修改 hive-site.xml 文件的内容，由于 hive-site.xml 中内容较多，我们需要在本地打开文件删除文件中的内容，单击右键桌面“Computer”→

“Filesystem” → “simple” → “hive-0.12.0” → “conf”，单击右键 hive-site.xml 文件选择 Open With gedit 进行编译，删除<configuration></configuration>中所有内容，此操作会比较耗时，操作完之后在终端执行命令 vim hive-site.xml 之后并查看内容。注意：MySQL URL 路径地址的 IP 地址根据本机情况进行修改，如图 9-8、图 9-9 所示。

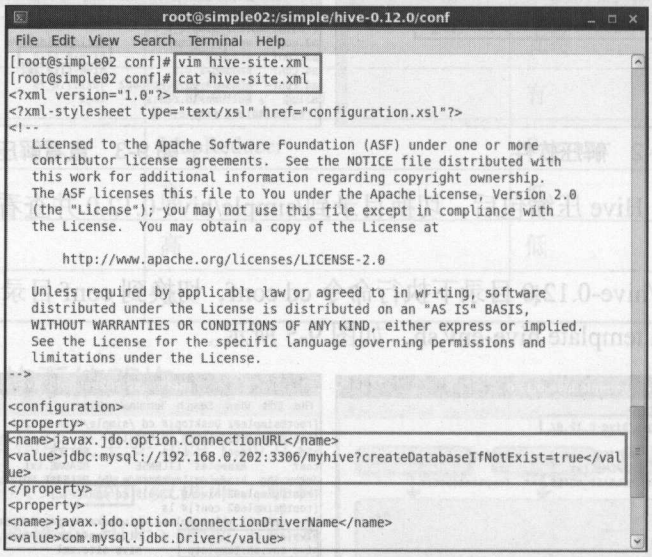



图 9-8　配置信息 1

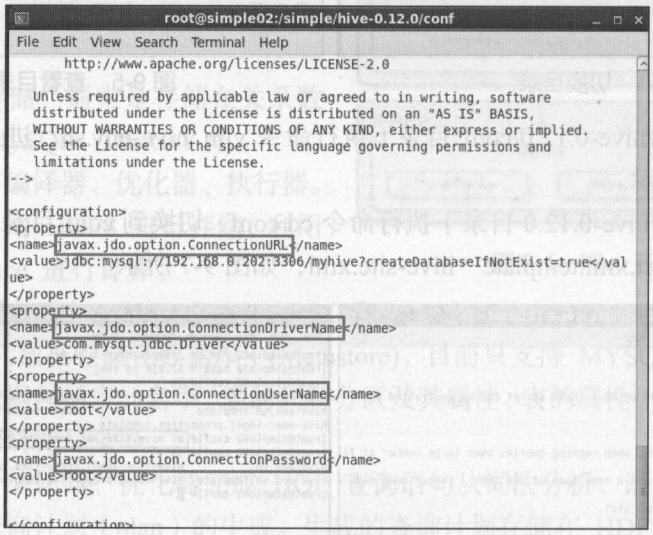



图 9-9　配置信息 2

（8）完成上一步之后，在目录$HIVE_HOME/bin 下面，修改文件 hive-config.sh，增加以下内容：

```
export JAVA_HOME=/simple/jdk1.7.0_79
export HIVE_HOME=/simple/hive-0.12.0
export HADOOP_HOME=/simple/hadoop-2.4.1。
```

如图 9-10、图 9-11 所示。

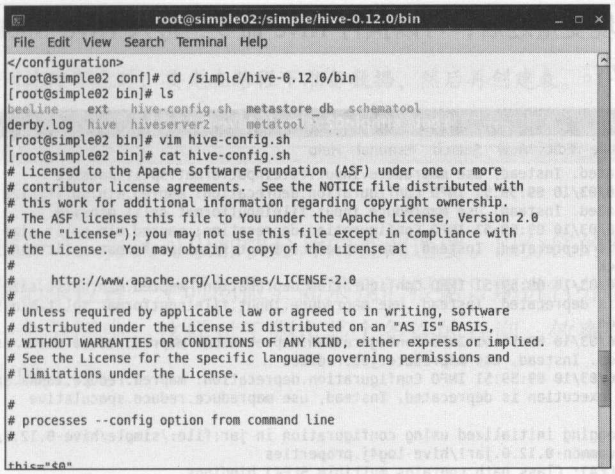



图 9-10 修改文件

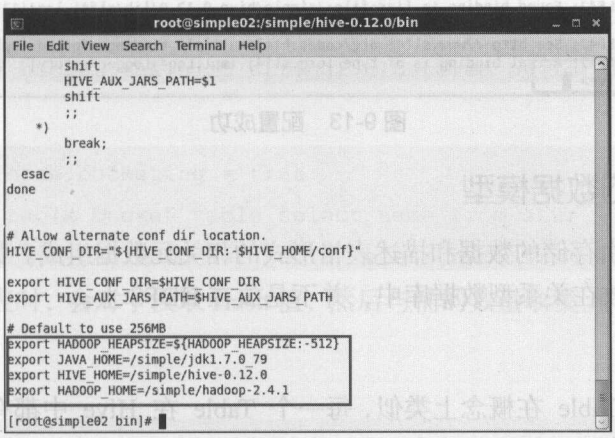



图 9-11 修改配置文件

（9）在命令终端任意目录下，执行命令 vim /etc/profile，然后编辑内容，进行 hive 环境变量的配置，如图 9-12 所示。

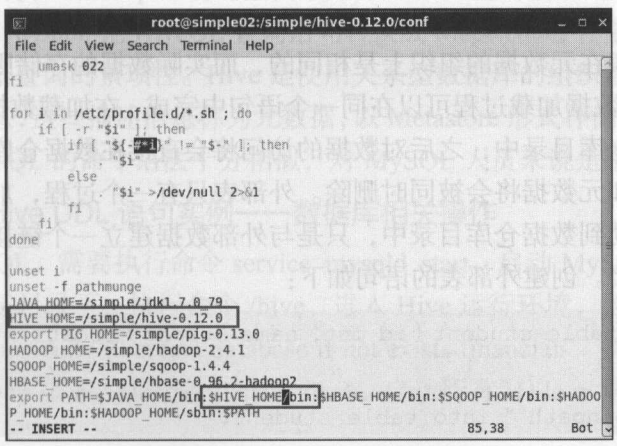



图 9-12 环境变量

（10）配置完环境变量之后，可以执行 Hive 命令，进入 Hive Shell 环境表示安装配置成功，如图 9-13 所示。

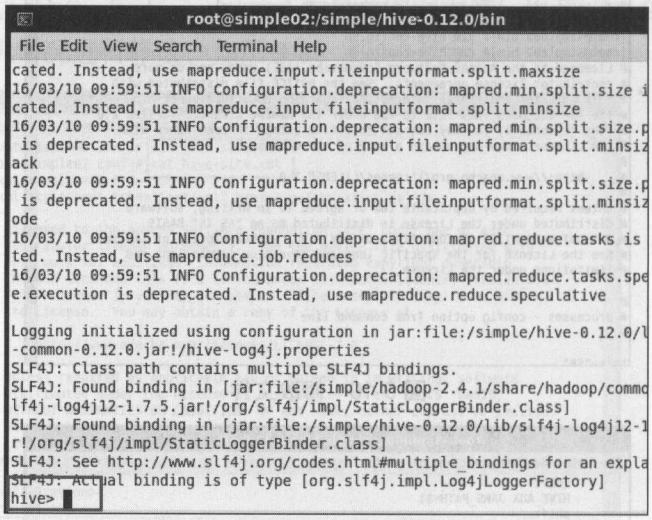



图 9-13　配置成功

## 9.3　Hive 的数据模型

Hive 表逻辑上由存储的数据和描述表格形式的相关元数据组成。数据一般放在 HDFS 上。Hive 把元数据放在关系型数据库中，并不是放在 HDFS 上。

### 9.3.1　内部表

与数据库中的 Table 在概念上类似，每一个 Table 在 Hive 中都有一个相应的目录存储数据。例如，一个表 test，它在 HDFS 中的路径为/ warehouse/test。 warehouse 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir} 指定的数据仓库的目录，所有的 Table 数据（不包括 External Table）都保存在这个目录中。删除表时，元数据与数据都会被删除。

### 9.3.2　外部表

外部表和内部表在元数据的组织上是相同的，而实际数据的存储则有较大的差异，内部表 的创建过程和数据加载过程可以在同一个语句中完成，在加载数据的过程中，实际数据会被移动到数据仓库目录中；之后对数据的访问将会直接在数据仓库目录中完成。删除表时，表中的数据和元数据将会被同时删除。外部表只有一个过程，加载数据和创建表同时完成，并不会移动到数据仓库目录中，只是与外部数据建立一个链接。当删除一个外部表时，仅删除该链接。创建外部表的语句如下：

```
Create external table student（id int, name string）location'路径'; //创建外部表时指定位置
Load data local inpath " into table student;
```

创建表时使用 external 关键字时，Hive 知道数据并不属于自己管理，不会把数据移到自己的数据仓库中。在丢失外部表时，只会删除元数据并不会删除数据信息。

说明

在创建外部表时，要先在路径下准备数据，然后再创建表。

### 9.3.3 分区表

Hive 把表组织成分区，分区可以加快分片的查询速度。以分区的常用情况为例。考虑日志文件，其中每条记录包含一个时间戳。如果我们根据日期进行分区，那么同一天的记录就会被存放在同一个分区中。这样对于限定某个条件的查询，效率会非常高，因为它只需要扫描查询范围内的文件。

### 9.3.4 桶表

桶表是对数据进行哈希取值，然后放到不同文件中存储。

创建表

```
create table bucket_table(id string) clustered by(id) into 4 buckets;
```

加载数据

```
set hive.enforce.bucketing = true;
insert into table bucket_table select name from stu;
insert overwrite table bucket_table select name from stu;
```

数据加载到桶表时，会对字段取 Hash 值，然后与桶的数量取模。把数据放到对应的文件中。

抽样查询

```
select * from bucket_table tablesample(bucket 1 out of 4 on id);
```

## 9.4 Hive Shell 操作

Hive Shell 是运行在 Hadoop 环境之上，是实现和 Hive 进行交互的命令行接口，当执行 HiveQL 命令时，Hive Shell 把 HiveQL 查询语句转换成 MapReduce 作业进行处理并返回结果，这样简化了数据查询的繁琐性。Hive 是使用关系型数据库的组织形式存储 HDFS 上的存储信息的描述信息，这些描述信息称为元数据，以 Metastore 形式存储在关系型数据库中。HiveQL 命令和 MySQL 的命令语法十分相似，对 MySQL 人员来说是个福音。

**【案例 9-2】Hive DDL 语句实例——数据库相关操作**

（1）启动 MySQL，需要执行命令 service mysqld start，启动 MySQL 服务。然后进入 Hive 的安装目录 bin 目录下，执行命令./hive，进入 Hive 运行环境，执行创建数据库的命令 create database financials 和 create database if not exists financials，执行这些命令之前，确保正常启动 Hadoop 服务，执行命令 start-all.sh，启动所有 Hadoop 服务进程，如图 9-14 所示。

```
16/07/05 13:28:05 INFO Configuration.deprecation: mapred.reduce.tasks.spe
ulative.execution is deprecated. Instead, use mapreduce.reduce.speculativ

Logging initialized using configuration in jar:file:/simple/hive-0.12.0/l
b/hive-common-0.12.0.jar!/hive-log4j.properties
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/simple/hadoop-2.4.1/share/hadoop/commo
/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/simple/hive-0.12.0/lib/slf4j-log4j12-1
6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an expla
ation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive> create database financials;
OK
Time taken: 3.754 seconds
hive> create database if not exists financials;
OK
Time taken: 0.09 seconds
hive>
```

图 9-14　创建数据库

（2）创建数据库之后，如果需要查看数据库是否创建成功，需要在 Hive Shell 环境下，执行命令 show databases 进行查看，如图 9-15 所示。

```
SLF4J: Found binding in [jar:file:/simple/hadoop-2.4.1/share/hadoop/com
/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/simple/hive-0.12.0/lib/slf4j-log4j12
6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an exp
ation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive> create database financials;
OK
Time taken: 3.754 seconds
hive> create database if not exists financials;
OK
Time taken: 0.09 seconds
hive> show databases;
OK
default
financials
simple
Time taken: 0.298 seconds, Fetched: 3 row(s)
hive>
```

图 9-15　查看数据库 1

（3）如果想查询指定的一些数据库，可以使用 LIKE 或者正则表达式查看数据库，查询以 f 开头的数据库，执行命令 show databases like 'f.* '，查询到 financials，如图 9-16 所示。

```
OK
Time taken: 3.754 seconds
hive> create database if not exists financials;
OK
Time taken: 0.09 seconds
hive> show databases;
OK
default
financials
simple
Time taken: 0.298 seconds, Fetched: 3 row(s)
hive> show databases like 'f.*;'
    > ;
FAILED: ParseException line 1:24 character '<EOF>' not supported here

hive> show databases like 'f.*';
OK
financials
Time taken: 0.016 seconds, Fetched: 1 row(s)
hive>
```

图 9-16　查看数据库 2

思考以上 like 后面的“f.*”表示什么意思。

（4）如果想查看已经存在的数据库的详细信息，可以执行命令 describe database financials，查看数据库的详细信息，如图 9-17 所示。

```
default
financials
simple
Time taken: 0.298 seconds, Fetched: 3 row(s)
hive> show databases like 'f.*;'
    > ;
FAILED: ParseException line 1:24 character '<EOF>' not supported here

hive> show databases like 'f.*';
OK
financials
Time taken: 0.016 seconds, Fetched: 1 row(s)
hive> describe databases financials;
FAILED: SemanticException [Error 10001]: Table not found databases
hive> describe database financials;
OK
financials              hdfs://host01:9000/user/hive/warehouse/financials
db
Time taken: 0.019 seconds, Fetched: 1 row(s)
hive>
```

图 9-17 数据库信息

从图中可以看出，数据库所在目录位于 hdfs://localhost:9000/user /hive/warehouse/financials.db 目录中。

（5）如果创建表的时候，希望修改数据库所在目录，可以执行命令 create database financials1 location '/user/hive/warehouse/test'，创建一个指定目录的位置的数据库，如图 9-18 所示。

```
db
Time taken: 0.019 seconds, Fetched: 1 row(s)
hive> create database financials1
    > location 'usr/local/hive/warehouse/test/';
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.
.DDLTask. MetaException(message:java.lang.IllegalArgumentException: ja
et.URISyntaxException: Relative path in absolute URI: hdfs://host01:90
usr/local/hive/warehouse/test)
hive> create database financials1 location '/user/hive/warehouse/test';
OK
Time taken: 0.036 seconds
hive> describe database financials1;
OK
financials1             hdfs://host01:9000/user/hive/warehouse/test
Time taken: 0.022 seconds, Fetched: 1 row(s)
hive>
```

图 9-18 修改数据库目录

（6）如果创建数据库时，希望给数据库增加描述信息，可以执行命令 create database financials3 comment 'helloworld',为数据库增加描述信息，如图 9-19 所示。

```
        at org.apache.hadoop.hive.cli.CliDriver.processCmd(CliDriver.java:216)
        at org.apache.hadoop.hive.cli.CliDriver.processLine(CliDriver.java:413)
        at org.apache.hadoop.hive.cli.CliDriver.executeDriver(CliDriver.java:781)
        at org.apache.hadoop.hive.cli.CliDriver.run(CliDriver.java:675)
        at org.apache.hadoop.hive.cli.CliDriver.main(CliDriver.java:614)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav
a:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessor
Impl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.apache.hadoop.util.RunJar.main(RunJar.java:212)
FAILED: ParseException line 1:14 cannot recognize input near ''financials2'' '<EOF>
' '<EOF>' in drop database statement

hive> create database financials3 comment 'helloworld';
OK
Time taken: 0.683 seconds
hive> describe database financials3;
OK
financials3     helloworld      hdfs://host01:9000/user/hive/warehouse/financials3.
db
Time taken: 0.02 seconds, Fetched: 1 row(s)
hive>
```

图 9-19 描述信息

（7）如果创建数据库时，需要为数据库增加属性信息，可以使用 with dbproperties 参数，执行命令

```
create database financials4
with dbproperties('creator'='hello','date'='2016-06-27');
```

为数据库增加属性信息，如图 9-20 所示。

```
hive> create database financials3 comment 'helloworld';
OK
Time taken: 0.683 seconds
hive> describe database financials3;
OK
financials3     helloworld      hdfs://host01:9000/user/hive/warehouse/financials3.
db
Time taken: 0.02 seconds, Fetched: 1 row(s)
hive> create database financials4 with dbproperties('creator'='hello','date'='2016-06-27');
OK
Time taken: 0.048 seconds
hive> describe database financials4;
OK
financials4             hdfs://host01:9000/user/hive/warehouse/financials4.db
Time taken: 0.024 seconds, Fetched: 1 row(s)
hive> describe database extended financials4;
OK
financials4             hdfs://host01:9000/user/hive/warehouse/financials4.db   {date=2016-06-2
7, creator=hello}
Time taken: 0.016 seconds, Fetched: 1 row(s)
hive>
```

图 9-20　增加属性

思考 extended 关键字添加与不添加的区别。

（8）如果要使用已存在的数据库，需要使用 use 关键字说明之后才可以，在 Hive Shell 环境下，执行命令 use financials3,表示可以使用该数据库了，执行命令 show tables,如图 9-21 所示。

```
db
Time taken: 0.02 seconds, Fetched: 1 row(s)
hive> create database financials4 with dbproperties('creator'='hello','
OK
Time taken: 0.048 seconds
hive> describe database financials4;
OK
financials4             hdfs://host01:9000/user/hive/warehouse/financia
Time taken: 0.024 seconds, Fetched: 1 row(s)
hive> describe database extended financials4;
OK
financials4             hdfs://host01:9000/user/hive/warehouse/financia
7, creator=hello}
Time taken: 0.016 seconds, Fetched: 1 row(s)
hive> use financials3;
OK
Time taken: 0.012 seconds
hive> show tables;
OK
Time taken: 0.139 seconds
hive>
```

图 9-21　使用数据库

（9）如果要修改已存在的数据库，在 Hive Shell 环境下，执行命令 alter database financials set dbproperties('edited-by'='hello');,修改数据库的属性信息，如图 9-22 所示。

```
        at org.apache.hadoop.hive.ql.Driver.runInternal(Driver.java:977)
        at org.apache.hadoop.hive.ql.Driver.run(Driver.java:888)
        at org.apache.hadoop.hive.cli.CliDriver.processLocalCmd(CliDriver.java
        at org.apache.hadoop.hive.cli.CliDriver.processCmd(CliDriver.java:216)
        at org.apache.hadoop.hive.cli.CliDriver.processLine(CliDriver.java:413
        at org.apache.hadoop.hive.cli.CliDriver.executeDriver(CliDriver.java:7
        at org.apache.hadoop.hive.cli.CliDriver.run(CliDriver.java:675)
        at org.apache.hadoop.hive.cli.CliDriver.main(CliDriver.java:614)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImp
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcc
)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.apache.hadoop.util.RunJar.main(RunJar.java:212)
FAILED: ParseException line 1:0 cannot recognize input near 'disable' ''financ

hive> alter database financials
    > set dbproperties('edited-by'='hello');
OK
Time taken: 0.067 seconds
hive>
```

图 9-22　修改数据库

注意

数据库的其他元数据信息都是不可更改的，没有办法可以删除或者重置数据库属性。

【案例 9-3】Hive DDL 语句实例——表相关操作

（1）在 Linux 命令终端，执行命令 service mysqld start,启动 MySQL 数据库服务。如果希望创建 Hive 表之前，确保启动 Hadoop 服务并进入 Hive Shell 环境，分别执行命令 start-all.sh 和在 hive 安装目录 bin 下，执行命令./hive，如图 9-23 所示。

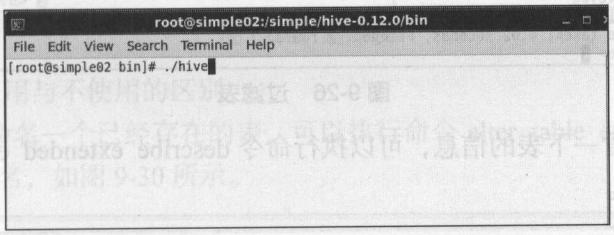



图 9-23 进入 Hive Shell 环境

（2）如果想通过已存在的表创建一个新的表结构，可以通过 Like 关键字拷贝已有的表来创建新表，执行命令 create table if not exists emp like employee1;，创建一个和 employee1 结构一致的表，如图 9-24 所示。

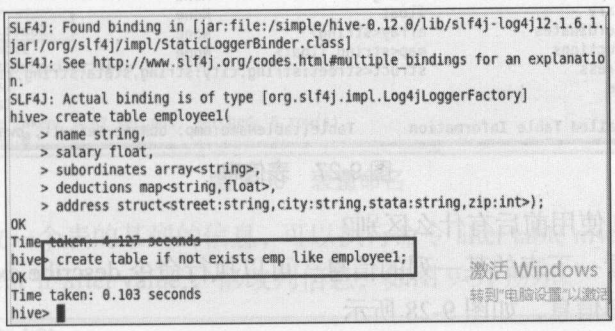



图 9-24 拷贝表

（3）在指定的数据库下，如果想查看该数据库中的所有表，可以执行命令 show tables，显示所有的表，如图 9-25 所示。

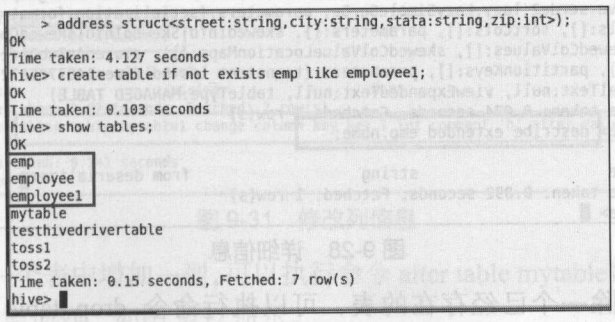



图 9-25 显示表

（4）在指定的数据库下，如果想查看该数据库中的指定的表，可以使用正则表达式过滤表，执行命令 show tables 'emp.* ',如图 9-26 所示。

```
Time taken: 0.103 seconds
hive> show tables;
OK
emp
employee
employee1
mytable
testhivedrivertable
toss1
toss2
Time taken: 0.15 seconds, Fetched: 7 row(s)
hive> show tables "emp.*";
OK
employee
employee1
Time taken: 0.02 seconds, Fetched: 2 row(s)
hive>
```

图 9-26　过滤表

（5）如果想查看一下表的信息，可以执行命令 describe extended emp,描述表的详细信息，如图 9-27 所示。

```
name                    string                  None
salary                  float                   None
subordinates            array<string>           None
deductions              map<string,float>       None
address                 struct<street:string,city:string,stata:string,zip:i
None
Time taken: 0.094 seconds, Fetched: 5 row(s)
hive> describe extended emp;
OK
name                    string                  None
salary                  float                   None
subordinates            array<string>           None
deductions              map<string,float>       None
address                 struct<street:string,city:string,stata:string,zip:i
None

Detailed Table Information      Table(tableName:emp, dbName:default, owner:
```

图 9-27　表信息

考虑：extended 使用前后有什么区别?

（6）如果想查看一下表的某一列的信息，可以执行命令 describe extended emp.name,描述表中某一列的详细信息，如图 9-28 所示。

```
g>, comment:null), FieldSchema(name:deductions, type:map<string,float>,
t:null), FieldSchema(name:address, type:struct<street:string,city:strin
:string,zip:int>, comment:null)], location:hdfs://host01:9000/user/hive/
use/emp, inputFormat:org.apache.hadoop.mapred.TextInputFormat, outputFo
g.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat, compressed:fa
mBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org.apache
.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.format=1}),
tCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNam
 skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirecto
lse), partitionKeys:[], parameters:{transient_lastDdlTime=1467705672}, v
ginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE)
Time taken: 0.074 seconds, Fetched: 7 row(s)
hive> describe extended emp.name;
OK
name                    string                  from deserializer
Time taken: 0.092 seconds, Fetched: 1 row(s)
hive>
```

图 9-28　详细信息

（7）如果想删除一个已经存在的表，可以执行命令 drop table employee，删除表 employee，如图 9-29 所示。

```
hive> show tables;
OK
emp
employee
employee1
mytable
testhivedrivertable
toss1
toss2
Time taken: 0.012 seconds, Fetched: 7 row(s)
hive> drop table if exists emp;
OK
Time taken: 0.511 seconds
hive> drop table employee;
OK
Time taken: 0.113 seconds
hive>
```

图 9-29 删除表

考虑 if exists 使用与不使用的区别。

（8）如果想重命名一个已经存在的表，可以执行命令 alter table employee1 rename to employee11;,表重命名，如图 9-30 所示。

```
OK
employee1
mytable
testhivedrivertable
toss1
toss2
Time taken: 0.02 seconds, Fetched: 5 row(s)
hive> alter table employee1 rename to employee11;
OK
Time taken: 0.116 seconds
hive> show tables;
OK
employee11
mytable
testhivedrivertable
toss1
toss2
Time taken: 0.02 seconds, Fetched: 5 row(s)
hive>
```

图 9-30 表重命名

（9）如果想修改一个表的某列的信息，可以执行命令 alter table mytable1 change column key key_1 int comment 'h' after value;，修改列信息，如图 9-31 所示。

```
olumn type

hive> describe mytable;
OK
value                   string                  None
key_1                   int                     hw
Time taken: 0.069 seconds, Fetched: 2 row(s)
hive> create table mytable1(key int,value string) comment 'hw';
OK
Time taken: 0.041 seconds
hive> describe mytable1;
OK
key                     int                     None
value                   string                  None
Time taken: 0.07 seconds, Fetched: 2 row(s)
hive> alter table mytable1 change column key key_1 int comment 'h' after value;
OK
Time taken: 0.141 seconds
hive>
```

图 9-31 修改列信息

（10）如果想在一个表中增加一列，可以执行命令 alter table mytable1 add columns(value1 string,value2 string);,增加列，如图 9-32 所示。

```
hive> describe mytable1;
OK
value                   string                  None
key_1                   int                     h
Time taken: 0.071 seconds, Fetched: 2 row(s)
hive> alter table mytable1 add columns(value1 string,value2 string);
OK
Time taken: 0.167 seconds
hive> describe mytables1;
FAILED: SemanticException [Error 10001]: Table not found mytables1
hive> describe mytable1;
OK
value                   string                  None
key_1                   int                     h
value1                  string                  None
value2                  string                  None
Time taken: 0.058 seconds, Fetched: 4 row(s)
hive>
```

图 9-32　增加列

（11）如果想替换某个表中的列，可以执行命令 alter table mytable1 replace columns (value1 string，value11 string);,替换列，如图 9-33 所示。

```
hive> alter table mytable1 add columns(value1 string,value2 string);
OK
Time taken: 0.167 seconds
hive> describe mytables1;
FAILED: SemanticException [Error 10001]: Table not found mytables1
hive> describe mytable1;
OK
value                   string                  None
key_1                   int                     h
value1                  string                  None
value2                  string                  None
Time taken: 0.058 seconds, Fetched: 4 row(s)
hive> alter table mytable1 replace columns(value1 string,value11 string);
OK
Time taken: 0.127 seconds
hive> describe mytable1;
OK
value1                  string                  None
value11                 string                  None
Time taken: 0.068 seconds, Fetched: 2 row(s)
hive>
```

图 9-33　删除或者替换列

（12）如果想修改指定表中的属性，可以执行命令 alter table mytable1 set tblproperties('value1'='hello');,修改表属性，如图 9-34 所示。

```
hive> alter table mytable1 set tblproperties('value1'='hello');
OK
Time taken: 0.085 seconds
hive> describe extended mytable1;
OK
value1                  string                  None
value11                 string                  None

Detailed Table Information      Table(tableName:mytable1, dbName:default, owner:root, cre
ateTime:1467707736, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema
(name:value1, type:string, comment:null), FieldSchema(name:value11, type:string, comment:
null)], location:hdfs://host01:9000/user/hive/warehouse/mytable1, inputFormat:org.apache.
hadoop.mapred.TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTex
tOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializat
ionLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.form
at=1}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[
], skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirectories:false), par
titionKeys:[], parameters:{last_modified_by=root, last_modified_time=1467709052, transien
t_lastDdlTime=1467709052, comment=hw, value1=hello}, viewOriginalText:null, viewExpandedT
ext:null, tableType:MANAGED_TABLE)
Time taken: 0.077 seconds, Fetched: 4 row(s)
hive>
```

图 9-34　修改表属性

## 9.5　Hive API 操作

实现和 Hive 交互方式，Hive 提供了三种用户接口：CLI、JDBC/ODBC 和 WebUI。

- CLI，即 Shell 命令行。
- JDBC/ODBC 是 Hive 的 Java，与使用传统数据库 JDBC 的方式类似。
- WebUI 是通过浏览器访问 Hive。

第二种方式是基于 Hadoop 的 Hive 数据仓库 Java API 简单调用的实例。Hive 提供了 JDBC 驱动，可以让开发者用 Java 代码来连接 Hive 并进行一系列的操作，同关系型数据库类似，也要将 Hive 的服务器打开，然后进行连接，最后才能对 Hive 中的数据进行操作。

**【案例 9-4】Hive 的 JDBC 实例（MySQL 数据库）**

Hive 所需要的组件中只有一个外部组件是 Hadoop 没有的，那就是 Metastore(元数据存储)组件。元数据存储中存储了如表的模式和分区信息等元数据信息。用户在执行如 create table x…或者 alter table y…等命令时会指定这些信息。因为多用户和系统可能需要并发访问元数据存储，所以默认的内置数据库并不适用于生产环境。

任何一个适用 JDBC 进行连接的数据库都可用作元数据存储。在实践中，大多数的 Hive 客户端会使用 MySQL。我们也将讨论如何使用 MySQL 来进行元数据存储。执行过程对于其他适用于 JDBC 连接的数据库都是适用的。

（1）启动 MySQL,执行命令 service mysqld start,具体操作前面实验已经讲过。通过 start-all.sh 启动 Hadoop,通过 jps 查看是否启动成功，通过执行命令 cd /simple/hive-0.12.0/bin，开启端口监听用户的连接，执行命令 hive --service hiveserver2，启动 Hive 服务，如图 9-35 所示。

（2）当 Hive 的 MySQL 安装方式配置好了之后，在 Eclipse 中新建一个 Java 项目“HiveJdbcClient”，如图 9-36 所示。

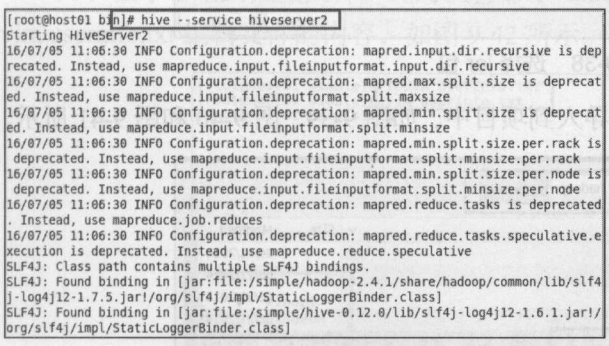



图 9-35 启动 MySQL

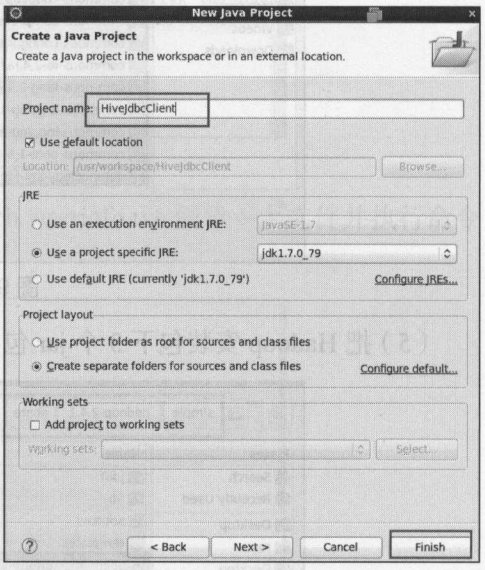



图 9-36 新建项目

（3）单击右键项目，选择“Build Path”→“Configure Build Path”进行导入包，如图 9-37 所示。

（4）把 Hive 安装目录下的 lib 目录下的 jar 包全部导入，如图 9-38 所示。

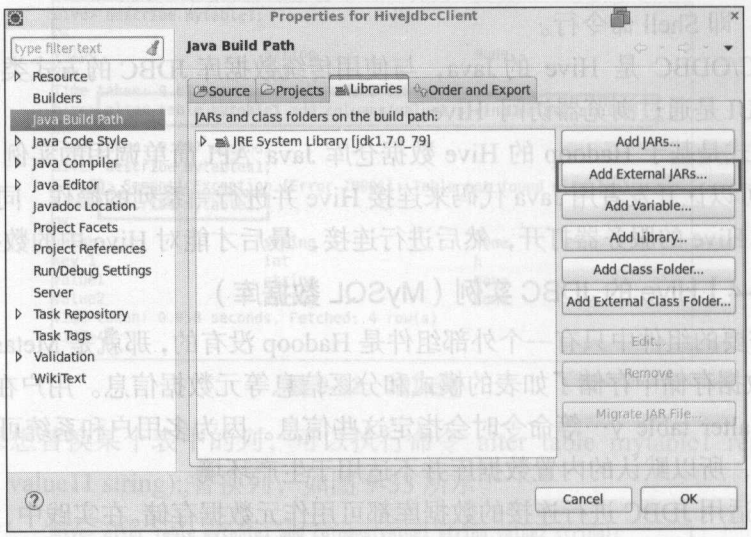



图 9-37　导入 jar 包

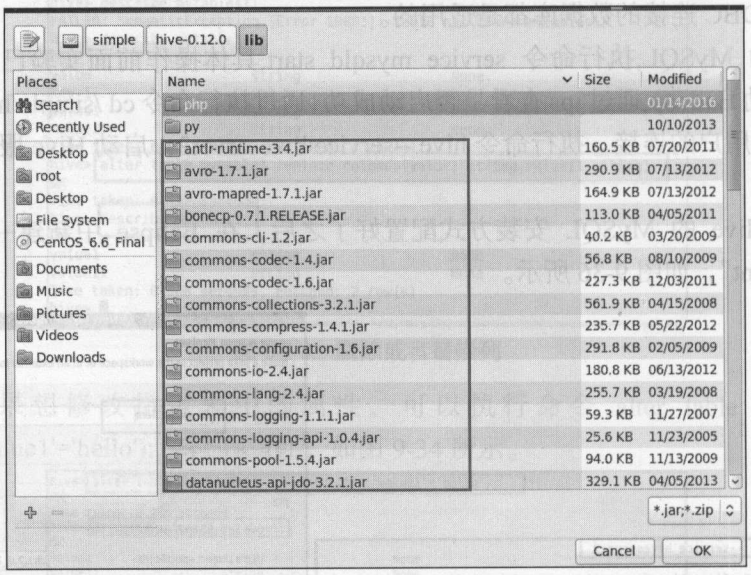



图 9-38　选择 jar 包

（5）把 Hadoop 安装包下 3 个 jar 包导入到项目中，如图 9-39、图 9-40 和图 9-41 所示。

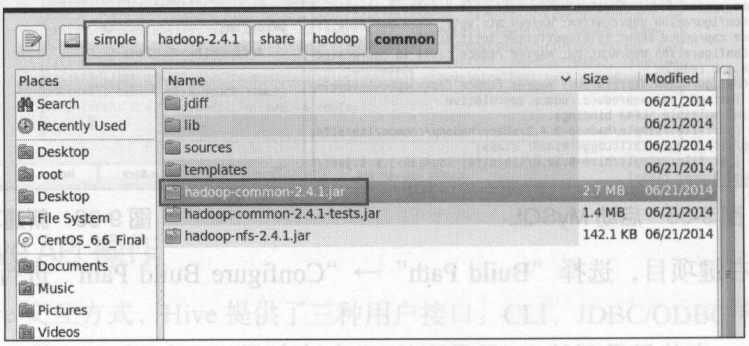



图 9-39　jar 包 1

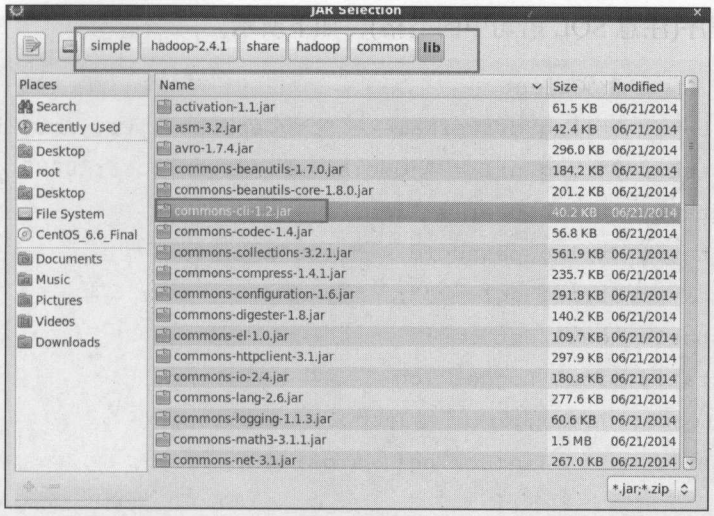



图 9-40 jar 包 2

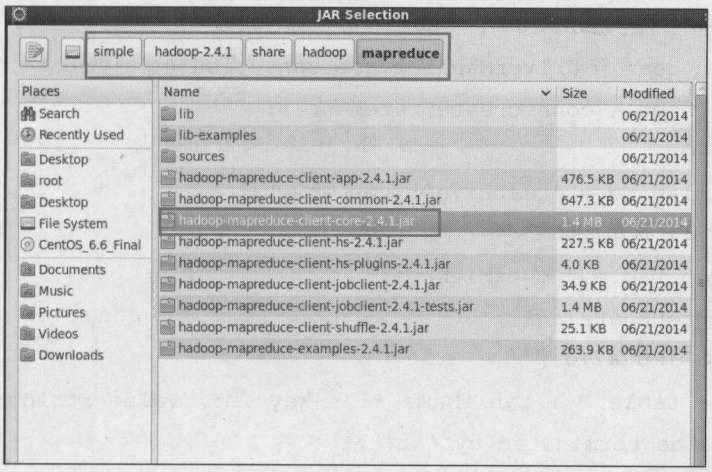



图 9-41 jar 包 3

（6）程序中用到数据文件，此时执行命令 touch userinfo.txt 创建一个文件并执行命令 vim userinfo.txt 编辑文件的内容，如图 9-42 所示。

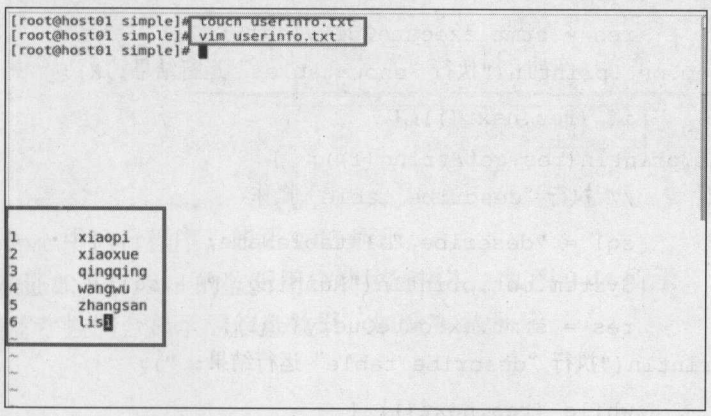



图 9-42 编译文件

（7）编写程序(注意 SQL 语句中的空格)，如下所示。

```
public class HiveJdbcClient {
    private static String driverName = "org.apache.hive.jdbc.HiveDriver";
    private static String url = "jdbc:hive2://localhost:10000/default";
    private static String user = "root";
    private static String password = "root";
    private static String sql = "";
    private static ResultSet res;
    private static final Logger log
           = Logger.getLogger(hiveJdbcClient.class);
    public static void main(String[] args) {
    try {
 Class.forName(driverName);
    //默认使用端口 10000，使用默认数据库，用户名密码默认
    Connection conn = DriverManager.getConnection(url, "root","root");
    Statement stmt = conn.createStatement();
    // 创建的表名
    String tableName = "testHiveDriverTable";
    /** 第一步:存在就先删除 **/
    sql = "drop table " + tableName;
    stmt.executeUpdate(sql);
    /** 第二步:不存在就创建 **/
sql = "create table " + tableName + " (key int, value string)  row format
delimited fields terminated by '\t'";
                stmt.executeUpdate(sql);
                // 执行“show tables”操作
                sql = "show tables '" + tableName + "'";
                System.out.println("Running: " + sql);
                res = stmt.executeQuery(sql);
        System.out.println("执行“show tables”运行结果: ");
                if (res.next()) {
    System.out.println(res.getString(1));  }
                // 执行“describe table”操作
                sql = "describe " + tableName;
                System.out.println("Running: " + sql);
                res = stmt.executeQuery(sql);
 System.out.println("执行“describe table”运行结果: ");
                while (res.next()) {
 System.out.println(res.getString(1) + "\t" + res.getString(2));  }
```

```
                // 执行 "load data into table" 操作
                String filepath = "/simple /userinfo.txt";
sql = "load data local inpath '" + filepath + "' into table "+ tableName;
                System.out.println("Running: " + sql);
                stmt.executeUpdate(sql);
                // 执行 "select * query" 操作
                sql = "select * from " + tableName;
                System.out.println("Running: " + sql);
                res = stmt.executeQuery(sql);
System.out.println("执行 "select * query" 运行结果: ");
   while(res.next()) {
System.out.println(res.getInt(1) + "\t" + res.getString(2));  }
                // 执行 "regular hive query" 操作
                sql = "select count(1) from " + tableName;
                System.out.println("Running: " + sql);
                res = stmt.executeQuery(sql);
   System.out.println("执行 "regular hive query" 运行结果: ");
                while (res.next()) {
                    System.out.println(res.getString(1));
                }
                conn.close();
                conn = null;
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
                log.error(driverName + " not found! ", e);
                System.exit(1);
            } catch (SQLException e) {
                e.printStackTrace();
                log.error("Connection error! ", e);
                System.exit(1);
            }
    }
}
```

（8）在 Eclipse 中运行程序，如图 9-43 所示。

（9）结果验证如下，Hive 的监听用户端执行情况，如图 9-44 所示。

（10）这是程序控制台显示的信息效果，如图 9-45 所示。

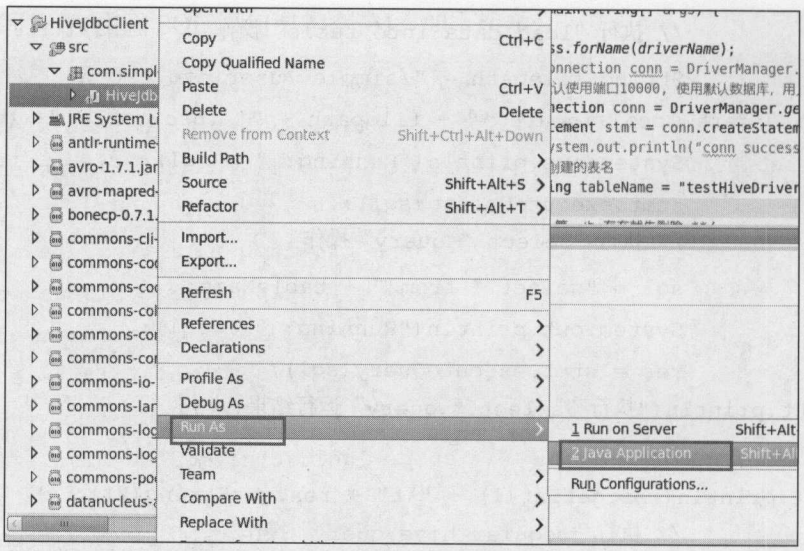



图 9-43　运行程序

```
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_1467682929926_0001, Tracking URL = http://host01:8088/proxy/appl
ication_1467682929926_0001/
Kill Command = /simple/hadoop-2.4.1/bin/hadoop job  -kill job_1467682929926_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2016-07-05 12:03:12,855 Stage-1 map = 0%,  reduce = 0%
2016-07-05 12:03:21,822 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:22,898 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:23,975 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:25,050 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:26,118 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:27,206 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.8 sec
2016-07-05 12:03:28,246 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 3.76 sec
2016-07-05 12:03:29,282 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 3.76 sec
MapReduce Total cumulative CPU time: 3 seconds 760 msec
Ended Job = job_1467682929926_0001
MapReduce Jobs Launched:
Job 0: Map: 1  Reduce: 1   Cumulative CPU: 3.76 sec   HDFS Read: 282 HDFS Write: 2
SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 760 msec
OK
```

图 9-44　执行效果

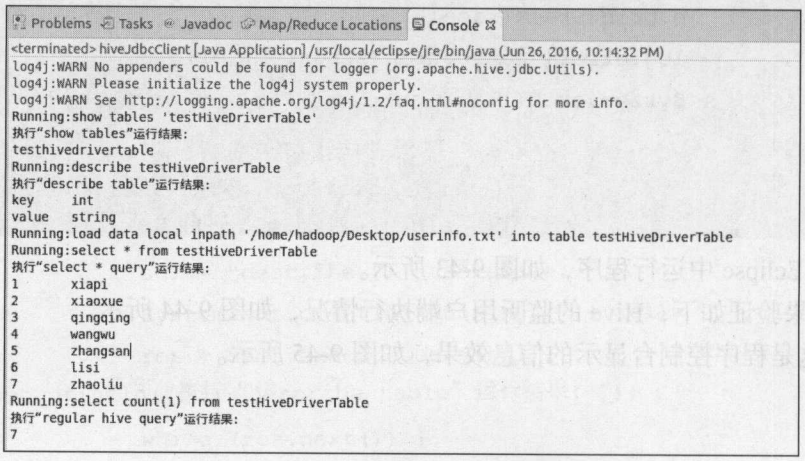



图 9-45　控制台显示信息

（11）重新开启另一终端，在 HDFS 上找到刚才上传的文件，如图 9-46 所示。

```
You have new mail in /var/spool/mail/root
[root@host01 simple]# hdfs dfs -ls /user
Found 1 items
drwxr-xr-x   - root supergroup          0 2016-06-28 18:03 /user/hive
[root@host01 simple]# hdfs dfs -ls /user/hive
Found 1 items
drwxr-xr-x   - root supergroup          0 2016-07-05 12:02 /user/hive/warehouse
[root@host01 simple]# hdfs dfs -ls /user/hive/warehouse
Found 5 items
drwxr-xr-x   - root supergroup          0 2016-07-05 10:38 /user/hive/warehouse/
mytable
drwxr-xr-x   - root supergroup          0 2016-06-28 18:05 /user/hive/warehouse/
simple.db
drwxr-xr-x   - root supergroup          0 2016-07-05 12:02 /user/hive/warehouse/
testhivedrivertable
drwxr-xr-x   - root supergroup          0 2016-07-05 10:25 /user/hive/warehouse/
toss1
drwxr-xr-x   - root supergroup          0 2016-07-05 10:26 /user/hive/warehouse/
toss2
[root@host01 simple]# hdfs dfs -ls /user/hive/warehouse/testhivedrivertable
Found 1 items
-rw-r--r--   1 root supergroup         57 2016-07-05 12:02 /user/hive/warehouse/
testhivedrivertable/userinfo.txt
[root@host01 simple]#
```

图 9-46　查看文件

## 9.6 Hive 内置函数和 UDF

### 9.6.1 内置函数

内置函数主要分为数学函数和聚合函数。

SHOWFUNCTIONS 命令可以列举出当前 Hive 会话中所加载的所有函数名称，其中包括内置的和用户加载进来的函数，加载方式稍后会进行介绍，如图 9-47 所示。

函数通常都有其自身的使用文档。使用 DESCRIBE FUNCTION 命令可以展示对应函数简短的介绍，如图 9-48 所示。

```
hive> SHOW FUNCTIONS;
abs
acos
and
array
```

图 9-47　显示函数

```
hive> DESCRIBE FUNCTION concat;
concat(str1, str2, ... strN) - returns the concatenation of str1, str2, ...
strN
```

图 9-48　函数介绍

### 9.6.2 UDF 函数

UDF 函数可以直接应用于 select 语句，对查询结构做格式化处理后，再输出内容。编写 UDF 函数的时候需要注意以下几点：

（1）自定义 UDF 需要继承 org.apache.hadoop.hive.ql.UDF。

（2）需要实现 evaluate 函数，evaluate 函数支持重载。

**【案例 9-5】Hive 通过日期计算星座实例**

（1）创建对应的表需要的数据文件 data.txt。在/simple 目录下执行命令 vim data.txt 并编辑内容，如图 9-49 所示。

（2）创建对应的表。在创建表时，先启动 MySQL，执行命令 service mysqld start,具体操作前面实验已经讲过。再确定 Hadoop 服务已经启动，否则执行 start-all.sh 命令，然后在 Hive 的安装目录 bin 目录下执行命令./hive 进入 Hive Shell 环境，执行创建表命令，如图 9-50 所示。

```
[root@host01 bin]# cd /simple
[root@host01 simple]# pwd
/simple
[root@host01 simple]# vim data.txt
[root@host01 simple]# cat data.txt
edward,edward@media6degrees.com,2-12-1981,209.191.139.200,M,10
bob,bob@test.net,10-10-2004,10.10.10.1,M,50
sara,sara@sky.net,4-5-1974,64.64.5.1,F,2
[root@host01 simple]#
```

图 9-49　切换目录

```
Time taken: 0.194 seconds, Fetched: 10 row(s)
hive> create table if not exists data(
    > name string,
    > email string,
    > bday string,
    > ip string,
    > gender string,
    > anum int)
    > row format delimited fields terminated by ',';
OK
Time taken: 0.953 seconds
hive>
```

图 9-50　创建表

（3）装载数据。在 Hive Shell 环境中，把前面创建的 data.txt 文件导入到表中，如图 9-51 所示。

```
hive> create table if not exists data(
    > name string,
    > email string,
    > bday string,
    > ip string,
    > gender string,
    > anum int)
    > row format delimited fields terminated by ',';
OK
Time taken: 0.953 seconds
hive> load data local inpath '/simple/data.txt' into table data;
Copying data from file:/simple/data.txt
Copying file: file:/simple/data.txt
Loading data to table default.data
Table default.data stats: [num_partitions: 0, num_files: 1, num_rows: 0, total_s
ize: 148, raw_data_size: 0]
OK
Time taken: 1.021 seconds
hive>
```

图 9-51　导入数据

（4）如果要查看指定的表中是否存在数据，可以使用查询语句查看数据表，执行命令 select * from data，查看 data 中的数据，如图 9-52 所示。

```
...
Time taken: 0.953 seconds
hive> load data local inpath '/simple/data.txt' into table data;
Copying data from file:/simple/data.txt
Copying file: file:/simple/data.txt
Loading data to table default.data
Table default.data stats: [num_partitions: 0, num_files: 1, num_rows: 0, total_s
ize: 148, raw_data_size: 0]
OK
Time taken: 1.021 seconds
hive> select * from data;
OK
edward  edward@media6degrees.com        2-12-1981       209.191.139.200 M       1
0
bob     bob@test.net    10-10-2004      10.10.10.1      M       50
sara    sara@sky.net    4-5-1974        64.64.5.1       F       2
Time taken: 0.282 seconds, Fetched: 3 row(s)
hive>
```

图 9-52　查看数据表

（5）在 Eclipse 中，创建一个项目名称“UDFDemo”，如图 9-53 所示。

（6）在 UDFDemo 项目的 src 目录下创建一个 Java 文件“UDFDemo”，如图 9-54 所示。

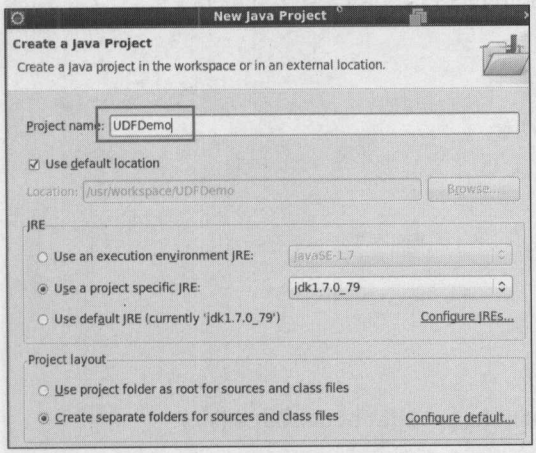



图 9-53 新建项目

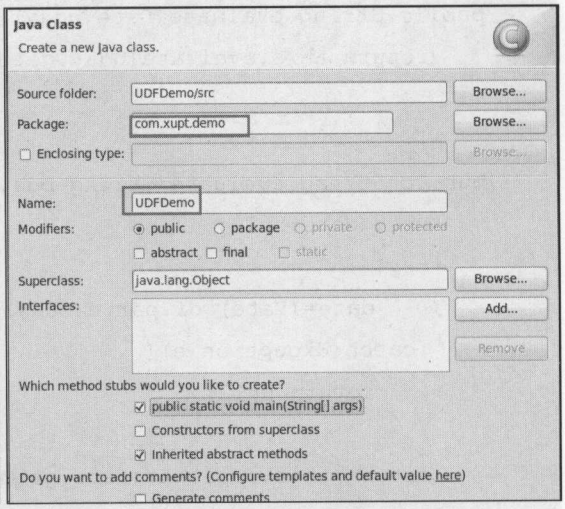



图 9-54 新建类

（7）在 UDFDemo 项目中，把/simple/hive-0.12.0/lib 下的所有 jar 包配置到 Path 路径中，如图 9-55 所示。

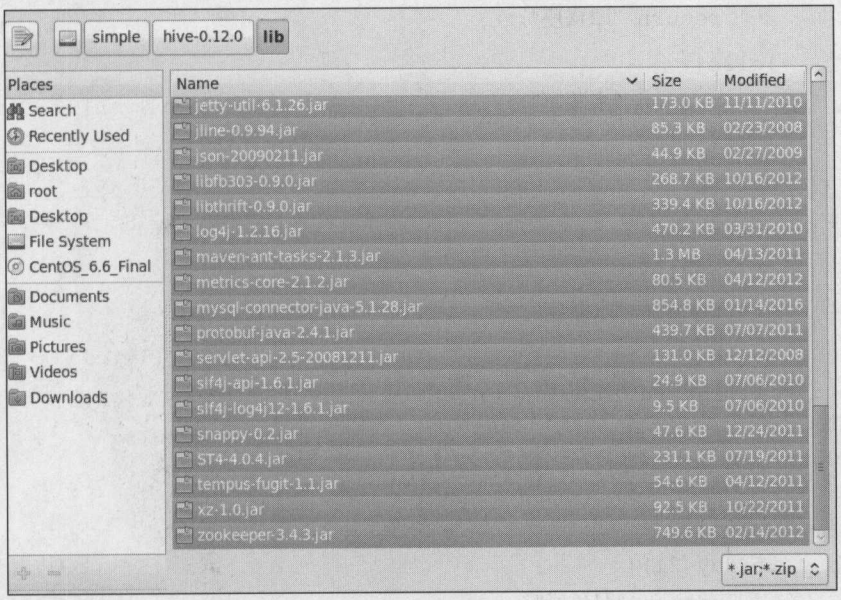



图 9-55 导入 jar 包 1

（8）编辑 UDFDemo.java 文件中的代码如下所示。

```
package com.xupt.demo;
public class UDFDemo extends UDF {
    private SimpleDateFormat df;
    public UDFDemo(){
        df=new SimpleDateFormat("MM-dd-yyyy");
```

```
    }
    public String evaluate(Date bday){
        return this.evaluate(bday.getMonth()+1,bday.getDate());
    }

    public String evaluate(String bday){
        Date date=null;
        try{
            date=(Date) df.parse(bday);
        }catch(Exception e){
            return null;
        }
        return this.evaluate(date.getMonth()+1,date.getDate());
    }
    public String evaluate(Integer month,Integer day){
        if(month==1){
            if(day<20){
                return "摩羯座";
            }else{
                return "水瓶座";
            }
        }
        if(month==2){
            if(day<19){
                return "水瓶座";
            }else{
                return "双鱼座";
            }
        }
        if(month==3){
            if(day<21){
                return "双鱼座";
            }else{
                return "白羊座";
            }
        }
        if(month==4){
            if(day<20){
                return "白羊座";
```

```
        }else{
            return "金牛座";
        }
    }
    if(month==5){
        if(day<21){
            return"金牛座";
        }else{
            return "双子座";
        }
    }
    if(month==6){
        if(day<22){
            return "双子座";
        }else{
            return "巨蟹座";
        }
    }
    if(month==7){
        if(day<23){
            return "巨蟹座";
        }else{
            return "狮子座";
        }
    }
    if(month==8){
        if(day<23){
            return "狮子座";
        }else{
            return "处女座";
        }
    }
    if(month==9){
        if(day<23){
            return "处女座";
        }else{
            return "天秤座";
        }
    }
```

```
        if(month==10){
            if(day<24){
                return "天秤座";
            }else{
                return "天蝎座";
            }
        }
        if(month==11){
            if(day<23){
                return "天蝎座";
            }else{
                return "射手座";
            }
        }
        if(month==12){
            if(day<22){
                return "射手座";
            }else{
                return"摩羯座";
            }
        }
        return null;
    }
}
```

（9）导出 jar 包，如图 9-56、图 9-57 和图 9-58 所示。

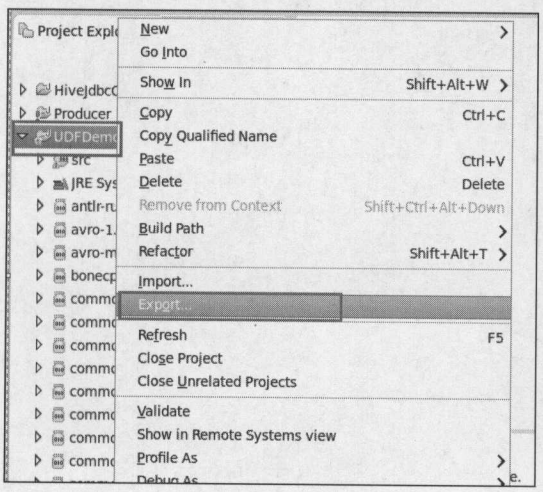



图 9-56　导出 jar 包 2

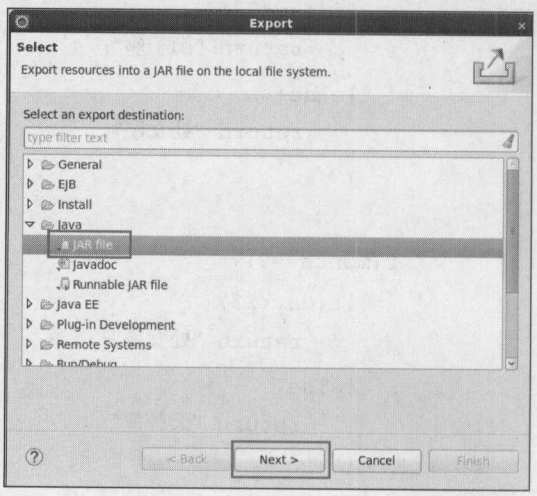



图 9-57　select

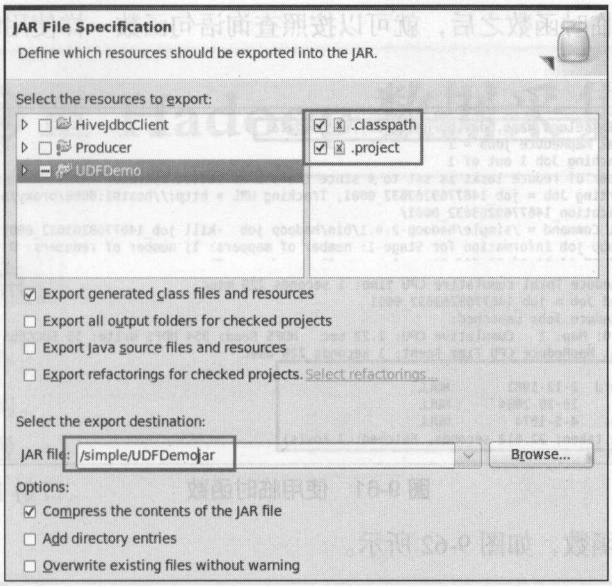



图 9-58　jar 位置

（10）执行命令 add jar /simple/UDFDemojar.jar;，添加 jar 文件到 Hive 环境，如图 9-59 所示。

```
c
2016-07-05 19:39:25,889 Stage-1 map = 100%,  reduce = 100%, Cumulativ
sec
2016-07-05 19:39:26,937 Stage-1 map = 100%,  reduce = 100%, Cumulativ
sec
MapReduce Total cumulative CPU time: 1 seconds 690 msec
Ended Job = job_1467713447993_0004
MapReduce Jobs Launched:
Job 0: Map: 1  Reduce: 1   Cumulative CPU: 1.69 sec   HDFS Read: 240
 2 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 690 msec
OK
3
Time taken: 20.252 seconds, Fetched: 1 row(s)
hive> add jar /simple/UDFDemojar.jar
    > ;
Added /simple/UDFDemojar.jar to class path
Added resource: /simple/UDFDemojar.jar
hive>
```

图 9-59　添加 jar 包

（11）要想使用程序中定义的函数，需要在 Hive 环境中定义临时函数，如图 9-60 所示。

```
ted. Instead, use mapreduce.job.reduces
16/07/06 09:41:46 INFO Configuration.deprecation: mapred.reduce.tasks.specul
e.execution is deprecated. Instead, use mapreduce.reduce.speculative

Logging initialized using configuration in jar:file:/simple/hive-0.12.0/lib/
-common-0.12.0.jar!/hive-log4j.properties
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/simple/hadoop-2.4.1/share/hadoop/common/l
lf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/simple/hive-0.12.0/lib/slf4j-log4j12-1.6.
r!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanat
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive> create temporary function UDFDemo as 'com.xupt.demo.UDFDemo';
FAILED: Class com.xupt.demo.UDFDemo not found
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.F
ionTask
hive> add jar /simple/UDFDemo.jar;
Added /simple/UDFDemo.jar to class path
Added resource: /simple/UDFDemo.jar
hive> create temporary function UDFDemo as 'com.xupt.demo.UDFDemo';
OK
Time taken: 0.022 seconds
hive>
```

图 9-60　临时函数

（12）创建完毕临时函数之后，就可以按照查询语句函数一样使用临时函数,如图 9-61 所示。

```
hive> select name,bday,UDFDemo(bday) from data;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1467769263632_0001, Tracking URL = http://host01:8088/proxy/a
pplication_1467769263632_0001/
Kill Command = /simple/hadoop-2.4.1/bin/hadoop job  -kill job_1467769263632_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0

MapReduce Total cumulative CPU time: 1 seconds 220 msec
Ended Job = job_1467769263632_0001
MapReduce Jobs Launched:
Job 0: Map: 1   Cumulative CPU: 1.22 sec   HDFS Read: 354 HDFS Write: 55 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 220 msec
OK
edward  2-12-1981       NULL
bob     10-10-2004      NULL
sara    4-5-1974        NULL
Time taken: 22.612 seconds, Fetched: 3 row(s)
hive>
```

图 9-61　使用临时函数

（13）删除临时函数，如图 9-62 所示。

```
MapReduce Total cumulative CPU time: 1 seconds 220 msec
Ended Job = job_1467769263632_0001
MapReduce Jobs Launched:
Job 0: Map: 1   Cumulative CPU: 1.22 sec   HDFS Read: 354 HDFS Write: 55 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 220 msec
OK
edward  2-12-1981       NULL
bob     10-10-2004      NULL
sara    4-5-1974        NULL
Time taken: 22.612 seconds, Fetched: 3 row(s)
hive> drop temporary function UDFDemo;
OK
Time taken: 0.005 seconds
hive>
```

图 9-62　删除临时函数

## 本章小结

在本章中，首先介绍了 Hive 的概念，然后介绍了 Hive 的数据模型，从几个常用的表进行介绍，分别说明内部表、外部表、分区表、桶表的使用，最后介绍 Hive Shell 和 Hive API 及 Hive 内置函数和 UDF 的使用，让学习者通过命令和 API 接口与 Hive 进行交互，解决数据存储问题。

## 习题

1. 简述 Hive 作为数据仓库的意义。
2. 简述 Hive 数据模型。
3. 理解 Hive 和 Hbase 的异同。

扫一扫在线测

# 第10章 Hadoop数据采集Flume

## 本章要点

- Flume 简介。
- Flume 架构。
- Flume 配置。
- Flume API 接口。

## 引　言

Flume 是 Cloudera 提供的高可用、高可靠、分布式的海量数据收集系统，从多种源数据系统采集、聚集和移动大量的数据并集中存储。当前 Flume 有两个版本，Flume 0.9X 版本的统称 Flume-OG，Flume1.X 版本的统称 Flume-NG，主要用于解决数据收集问题。本章通过对 Flume 简介、架构、常见命令、编程接口的讲解，让学生深刻理解和运用 Flume 系统。

## 10.1 Flume 概述

### 10.1.1 Flume 简介

官方网址：http://flume.apache.org/。

Flume 作为 Cloudera 开发的实时日志收集系统，在企业中得到广泛的应用。Flume 初始的发行版本目前被统称为 Flume OG( original generation )，属于 Cloudera。但随着 Flume 功能的扩展，Flume OG 代码工程臃肿、核心组件设计不合理、核心配置不标准等缺点暴露出来，尤其是在 Flume OG 的最后一个发行版本 0.94.0 中，日志传输不稳定的现象尤为严重。为了解决这些问题，2011 年 10 月 22 号，Cloudera 完成了 Flume-728，对 Flume 进行了里程碑式的改动，重构核心组件、核心配置以及代码架构，重构后的版本统称为 Flume NG ( next generation )；改动的另一原因是将 Flume 纳入 Apache 旗下，Cloudera Flume 改名为 Apache Flume。

### 10.1.2 Flume 核心概念

Flume 采用了分层架构，其中涉及很多概念，例如 Agent、Client、Source、Sink、Channel、Events 等。如表 10-1 所示。

表 10-1　组件

| 组　件 | 功　能 |
| --- | --- |
| Agent | 使用 JVM 运行 Flume，每台机器运行一个 agent，但是可以在一个 agent 中包含多个 sources 和 sinks |
| Client | 生产数据，运行在一个独立的线程 |
| Source | 从 Client 收集数据，传递给 Channel |
| Sink | 从 Channel 收集数据，运行在一个独立线程 |
| Channel | 连接 sources 和 sinks ，这个有点像一个队列 |
| Events | 可以是日志记录、 AVRO 对象等 |

### 10.1.3　Flume 系统要求

Java 运行环境 1.6 或 1.7 以上，建议使用 JDK1.7。

保证足够的内存用于配置使用的 sources、channels、sinks。

保证足够的磁盘空间用于配置使用的 sources、channels、sinks。

保证被 agent 使用的目录具有读写权限。

## 10.2　Flume 架构

Flume 架构整体上是 Source→Channel→Sink 的三层架构。

Source:完成对日志数据的收集，分成 Transition 和 Event 送入到 Channel 之中。

Channel:主要提供一个队列的功能，对 Source 中提供的数据进行简单的缓存。

Sink:取出 Channel 中的数据，进入相应的存储文件系统、数据库，或者提交到远程服务器。

Flume 以 Agent 为最小的独立运行单位。一个 Agent 就是一个 JVM。单 Agent 由 Source、Sink 和 Channel 三大组件构成，如图 10-1 所示。

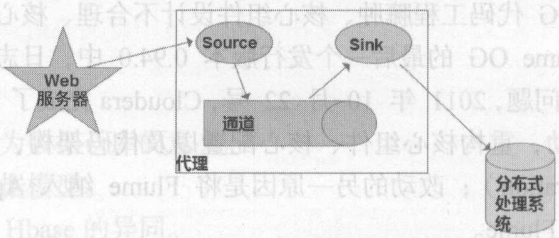



图 10-1　Flume 架构

Flume 的数据流由事件(Event)贯穿始终。事件是 Flume 的基本数据单位，它携带日志数据(字节数组形式)并且携带有头信息，这些 Event 由 Agent 外部的 Source，比如图 10-1 中的 Web 服务器生成。当 Source 捕获事件后会进行特定的格式化，然后 Source 会把事件推入(单个或多个)Channel 中。可以把 Channel 看作是一个缓冲区，它将保存事件直到 Sink 处理完该事件。Sink 负责持久化日志或者把事件推向另一个 Source。

Flume 提供了大量内置的 Source、Channel 和 Sink 类型。不同类型的 Source、Channel 和 Sink 可以自由组合。组合方式基于用户设置的配置文件，非常灵活。比如 Channel 可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink 可以把日志写入 HDFS、Hbase，甚至是另外一个 Source 等。

Flume 也可以支持用户建立多级流，也就是说，多个 Agent 可以协同工作，并且支持 Fan-in、Fan-out、Contextual Routing、Backup Routes。如图 10-2 所示。

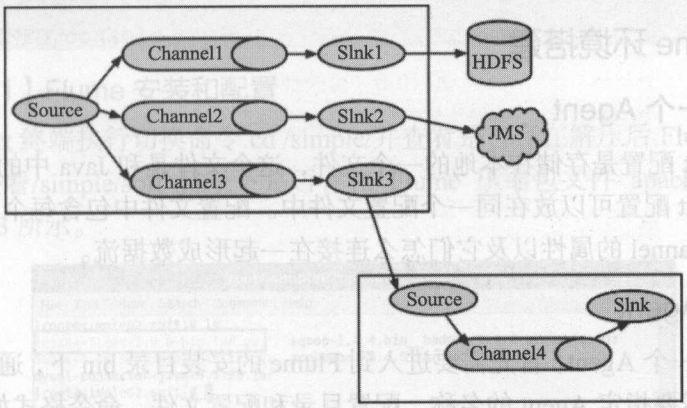



图 10-2 Flume 数据流

## 10.3 Flume 常见操作命令

```
# flume-ng 指令
commands:
  help                          显示帮助文本
  agent                         运行一个 Flume 代理
  avro-client                   运行一个 Avro 的 Flume 客户端
  version                       展示 Flume 的版本信息
global options:
  --conf,-c <conf>              使用 <conf> 目录下的配置
  --classpath,-C <cp>           添加 classpath
  --dryrun,-d                   并没有开始 Flume,只是打印命令
  -Dproperty=value              设置一个 Java 系统属性的值
  -Xproperty=value              设置一个 Java -x 选项

agent options:
  --name,-n <name>              这个 Agent 的名称(必需)
  --conf-file,-f <file>         指定一个配置文件 (如果有 -z 可以缺失)
  --zkConnString,-z <str>       指定使用的 Zookeeper 的链接 (如果有 -f 可以缺失)
  --zkBasePath,-p <path>        指定 agent config 在 Zookeeper Base Path
  --no-reload-conf              如果改变不重新加载配置文件
```

```
  --help,-h                显示帮助文本
avro-client options:
  --rpcProps,-P <file>     远程客户端与服务器链接参数的属性文件
  --headerFile,-R <file>   每个新的一行数据都会有的头信息 key/value
  --help,-h                显示帮助文本
其中--rpcProps 或 --host 和 --port 必须指定一个。
```

## 10.4 Flume 环境搭建

### 10.4.1 设置一个 Agent

Flume Agent 配置是存储在本地的一个文件，这个文件是和 Java 中的属性文件一致。一个或多个 Agent 配置可以放在同一个配置文件中。配置文件中包含每个 Source 的属性、Sink 的属性、Channel 的属性以及它们怎么连接在一起形成数据流。

### 10.4.2 启动 Agent

如果要启动一个 Agent，首先需要进入到 Flume 的安装目录 bin 下，通过执行脚本命令 flume-ng，此时需要指定 Agent 的名称、配置目录和配置文件。命令格式如下：

```
$bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.
template
```

配置文件内容：

```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = logger

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
```

```
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

启动命令：

```
$bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.
root.logger=INFO,console
```

【案例 10-1】Flume 安装和配置

（1）在 Linux 终端执行切换命令 cd /simple/并查看是否存在解压后 Flume 文件，如果不存在，则需要查看/simple/soft 目录下是否存在 Flume 压缩包文件 apache-flume-1.5.0-bin.tar.gz，如图 10-3 所示。

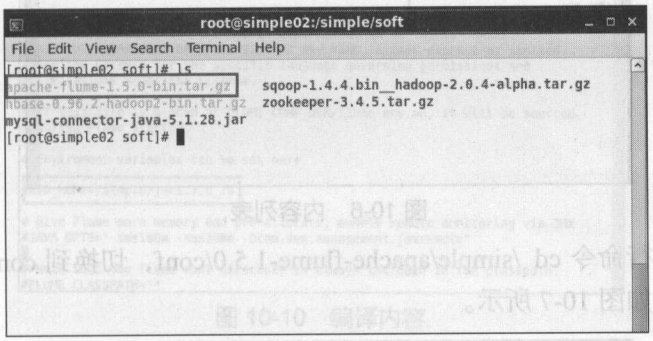



图 10-3 软件位置

（2）在 Linux 命令终端执行命令 cd /simple 切换到 simple 目录下，执行解压命令 tar-zxvf/simple/soft/apache-flume-1.5.0-bin.tar.gz 对 Flume 压缩包进行解压，如图 10-4 所示。

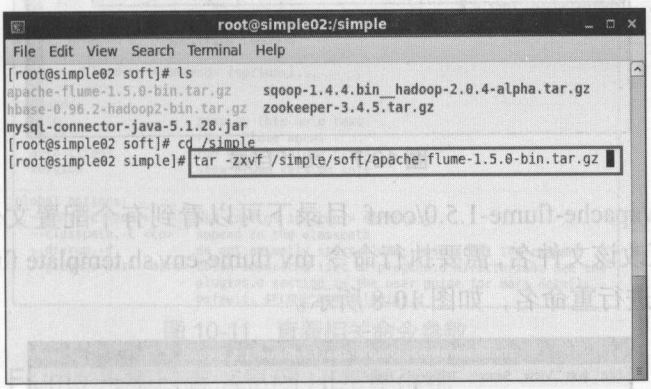



图 10-4 解压软件

（3）解压 Flume 压缩包之后，通过执行 ls 命令 ls /simple/apache-flume-1.5.0/来查看 Flume 解压后的目录结构，如图 10-5 所示。

（4）查看 Flume 目录下的 bin 目录下的内容列表。执行命令 ls/simple/apache-flume-1.5.0/bin 可以看到该目录下有一个执行文件 flume-ng，如图 10-6 所示。

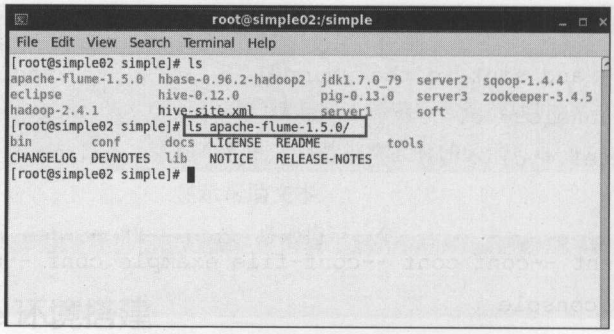



图 10-5　目录结构

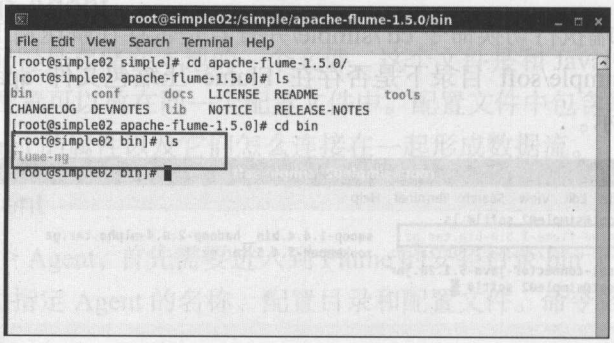



图 10-6　内容列表

（5）在终端执行命令 cd /simple/apache-flume-1.5.0/conf，切换到 conf 目录下，查看该目录下文件列表，如图 10-7 所示。

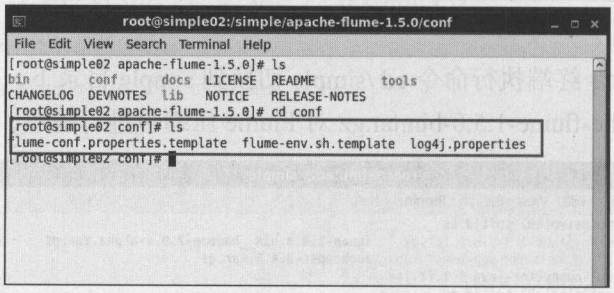



图 10-7　conf 目录

（6）在/simple/apache-flume-1.5.0/conf 目录下可以看到有个配置文件模板 flume-env.sh.template，首先更改该文件名，需要执行命令 mv flume-env.sh.template flume-env.sh 对 conf 目录下的配置文件进行重命名，如图 10-8 所示。

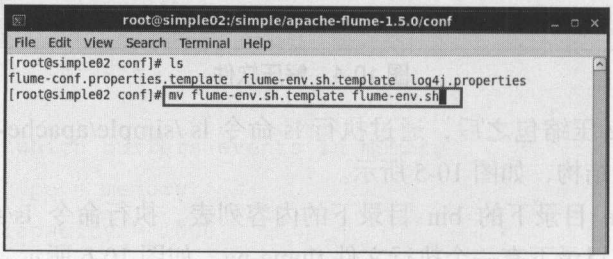



图 10-8　文件重命名

（7）首先通过编辑命令 vi flume-env.sh。然后修改配置文件中的内容，如图 10-9、图 10-10 所示。

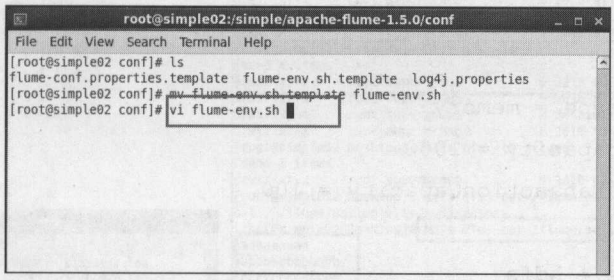



图 10-9 编译配置文件

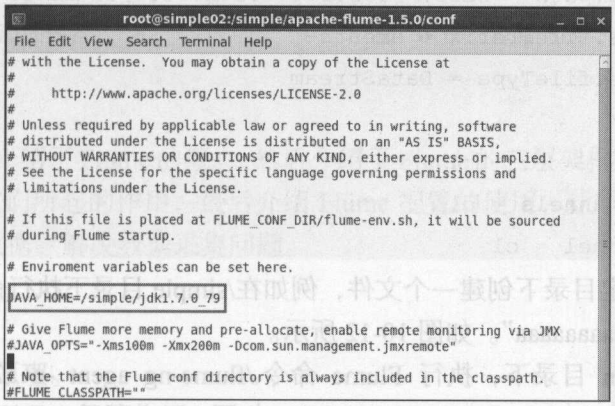



图 10-10 编译内容

（8）配置完毕之后切换到 Flume 目录下 bin 目录，执行命令./flume-ng 查看相关命令参数，如图 10-11 所示。

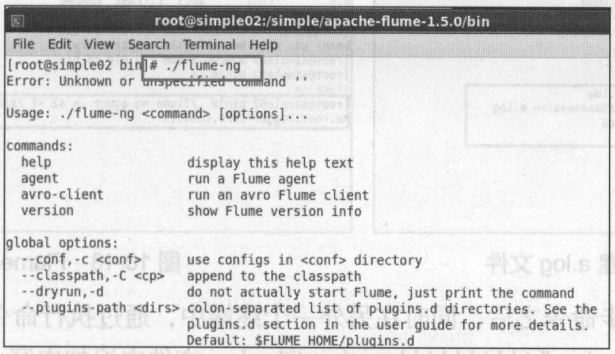



图 10-11 查看相关命令参数

【案例 10-2】Flume 采集日志信息到 HDFS 系统

（1）在任意指定目录下创建一个文件，例如在/simple 目录下执行命令 touch a2.conf 并向文件中写入内容如下

```
a2.sources = r1
a2.channels = c1
a2.sinks = k1
```

```
a2.sources.r1.type = exec
a2.sources.r1.command = tail -F /simple/a.log

a2.channels.c1.type = memory
a2.channels.c1.capacity = 1000
a2.channels.c1.transactionCapacity = 100

a2.sinks.k1.type = hdfs
a2.sinks.k1.hdfs.path = hdfs://192.168.0.202:9000/flume/aa.log
a2.sinks.k1.hdfs.filePrefix = events-
a2.sinks.k1.hdfs.fileType = DataStream


a2.sources.r1.channels = c1
a2.sinks.k1.channel = c1
```

（2）在任意指定目录下创建一个文件，例如在/simple 目录下执行命令 touch a.log 并向文件中写入内容“aaaaaaaaa”。如图 10-12 所示。

（3）切换到 bin 目录下，执行 Flume 命令./flume-ng agent -n a2 -f /simple/a2.conf -c ../conf/ -Dflume.root.logger=INFO,console。如图 10-13 所示。

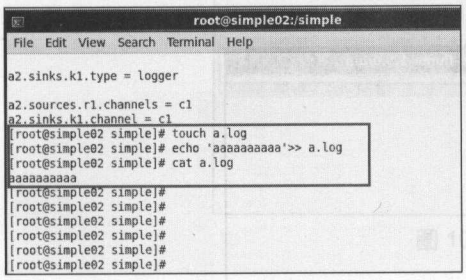



图 10-12　创建 a.log 文件

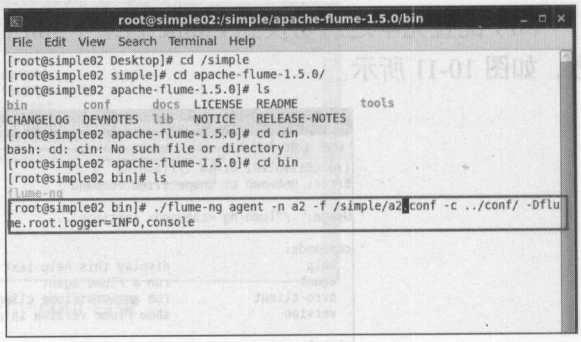



图 10-13　Flume 命令

（4）执行上一步命令之后，同时在另外一个终端中，通过执行命令 start-all.sh 启动服务，通过执行命令 echo 'bbbbbbbbbb'>>a.log 向 a.log 文件中追加内容。然后可以通过执行 HDFS 系统的命令查看 HDFS 中生成的文件并发现 HDFS 指定的目录文件下的内容增多。如图 10-14、图 10-15 和图 10-16 所示。

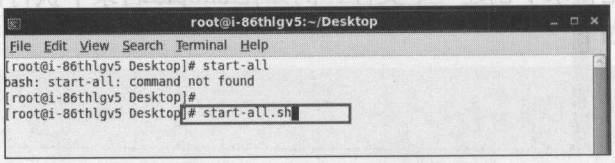



图 10-14　启动 Hadoop 服务

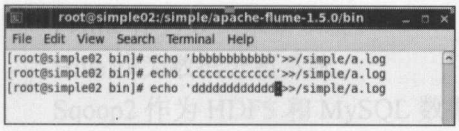



图 10-15 追加内容

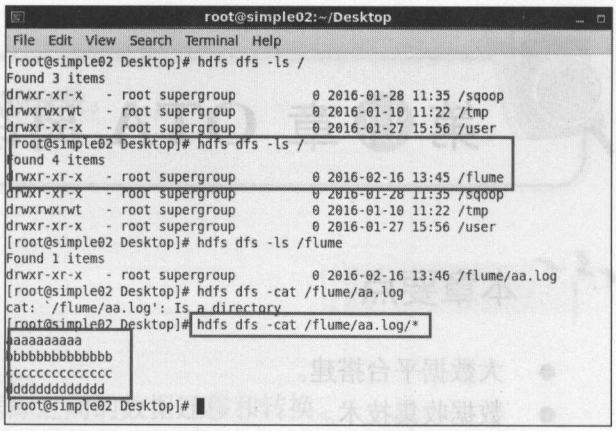



图 10-16 查看 HDFS 内容

## 本章小结

在本章中，首先介绍了 Flume 的概念，然后介绍了 Flume 的三层架构，从 source、channel、sink 三个概念讲解它们所起的作用，最后介绍 Flume 配置的使用，让学习者通过 Flume 工具从不同的源采集数据，解决数据采集问题。

## 习题

1. 简述 Flume 原理。
2. 简述 Flume 配置文件的作用。

扫一扫在线测

# 第11章 OTA 离线数据分析平台

## 本章要点

- 大数据平台搭建。
- 数据收集技术。
- 数据分析技术。
- 数据展示技术。

## 引　言

本章就是通过介绍如何开发一个基于 Hadoop 实现的离线数据平台，来理解大数据分析平台开发的原理及相关的技术知识点。让读者在实践中学会大数据相关技术的同时，能够利用现有技术解决现实中实际存在的问题。

## 11.1 项目概述

如今在线预定酒店已经成为用户的首选，其方便性及透明度均比过去电话预定提升了很多，而用户在线预定多会选择 OTA(Online Travel Agent)，即在线旅行社，代表有携程、艺龙、去哪。这些 OTA 在给用户带来方便的同时，也产生了很多用户访问量，并积累了很多用户数据。为了更好地提高酒店房间的入住率，间接提高自身收入，OTA 会分析数据的价值和它们存在的意义，那么选择一个大数据分析平台，看起来很有必要。

设计一款 OTA 大数据分析处理平台，对整个 OTA 行业提供了所需要的解决方案，能够解决类似各大 OTA 平台数量和交易金额纷纷大幅增长的问题。以携程和去哪为例，面对如此大的数据交易和自身数据的价值，开发一个合适的数据分析平台采集 OTA 数据、分析 OTA 数据、展示 OTA 数据很有必要。因为通过这样的大数据分析平台，才能了解客户，深入客户，才能更好地为客户服务。

离线分析平台方案主要分四层：存储平台层、数据日志数据采集层、数据分析处理层、数据展示层。

存储平台层负责提供数据存储、数据仓库的功能，包括 Hadoop、Hive 等。

数据采集层通过 Flume 把样本数据采集至 HDFS 中。

数据分析层分为 MapReduce 数据清洗、Hive 数据分析、Sqoop2 数据迁移。

数据展示层使用典型的 Web 框架把 MySQL 数据库中的数据以 ECharts 的曲线和分布图的形式展现出来。

## 11.2 功能需求

统计每家酒店未来一个月内每天的预定量。

所有酒店的预定区域分布。

## 11.3 软件开发关键技术

Hadoop 作为分布式计算平台。

Flume 作为数据采集系统。

Hive 作为数据分析系统。

Sqoop2 作为 HDFS 和 MySQL 数据库之间的数据迁移和转换。

ECharts 最终实现数据可视化。

## 11.4 效果展示

效果展示如图 11-1 所示。

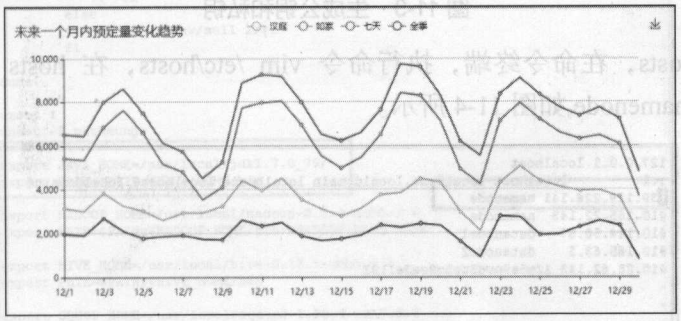



图 11-1 分析结果

## 11.5 平台搭建与测试

### 11.5.1 配置 ssh 免密码登录

（1）生成公钥密钥对。在命令终端执行命令 ssh-keygen，连续按 4 次回车，如图 11-2 所示。

```
[root@namenode ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
40:33:91:65:e6:8f:1a:88:87:6f:37:92:5a:e4:61:34 root@namenode
The key's randomart image is:
+--[ RSA 2048]----+
|      =++        |
|     ..*         |
|    E . .        |
|   + o . o       |
|  o * . S .      |
|   * o o         |
|    B +          |
|   + o .         |
|  .              |
+-----------------+
[root@namenode ~]#
```

图 11-2 生成公钥密钥对

（2）生成公钥和私钥。在/root/.ssh 目录下多出两个文件：私钥文件 id_raa 和公钥文件 id_rsa.pub，在命令终端，进入/root/.ssh 目录下，执行命令 cat id_rsa.pub >> authorized_keys，将公钥文件 id_rsa.pub 内容放到 authorized_keys 文件中,如图 11-3 所示。

```
[root@namenode ~]# cd /root/.ssh/
[root@namenode .ssh]# ll
total 8
-rw------- 1 root root 1675 Dec 28 11:44 id_rsa
-rw-r--r-- 1 root root  395 Dec 28 11:44 id_rsa.pub
[root@namenode .ssh]# cat id_rsa.pub >> authorized_keys
[root@namenode .ssh]# ll
total 12
-rw-r--r-- 1 root root  395 Dec 28 11:47 authorized_keys
-rw------- 1 root root 1675 Dec 28 11:44 id_rsa
-rw-r--r-- 1 root root  395 Dec 28 11:44 id_rsa.pub
[root@namenode .ssh]# cat authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAwN7LaygQiWJMkF+nw2Isjy1JIMCKz+x3oJsiHjz3J+ZC
QZpY6ttzk4yinwV735HN1U024iTF655PnU4lrIrWTQsipGDQl3/2SO8Mu/16KFeDCamvQxdtMXUck6C7
gTQLeTgRe0dh+KGNwATbzh9Vhin6iiDN87m4A0hMnaBCZrPFse+7WNx7/DK9zlamiVZ1N6EsEuCy0CbQ
XBOANrKLqBYS7i5DtDzwR+fXdExclqCrq8ekU+q5FvbfztUrLt+GRWtMjDTVRJz/dJssMXfcqYj+/u1H
LLTKE14yj4562eA2UckV+4+zvmcgrR9utBPQU/+kUGdGcQO15aOT8xx46Q== root@namenode
[root@namenode .ssh]#
```

图 11-3　生成公钥和私钥

（3）修改 hosts，在命令终端，执行命令 vim /etc/hosts，在 hosts 文件中添加信息 139.129.226.141 namenode,如图 11-4 所示。

```
127.0.0.1 localhost
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
139.129.226.141 namenode
#10.165.22.143  namenode
#10.144.58.57   datanode1
#10.165.63.5    datanode2
#10.28.82.143 iZm5e9mvd8tu5q9se3el19Z
~
~
~
~
~
~
~
```

图 11-4　主机名与 IP 对应

（4）在 namenode 节点的命令终端，执行命令 ssh namenode 验证 ssh 无密码登录,如图 11-5 所示。

```
[root@namenode ~]# ssh namenode
Last login: Wed Dec 28 11:50:36 2016 from 139.129.226.141

Welcome to aliyun Elastic Compute Service!

[root@namenode ~]#
```

图 11-5　验证 ssh 登录

## 11.5.2　配置 JDK

（1）将 JDK 解压至/usr/local，执行命令 tar -zxvf/usr/local/jdk-7u79-linux-x64.tar.gz -C /usr/local/，产生解压文件。如图 11-6 所示。

```
[root@namenode ~]# ll /usr/local/
total 68
drwxr-xr-x   6 root root 4096 Dec 17 13:42 aegis
drwxr-xr-x   8 1106 4001 4096 Dec 17 13:48 apache-flume-1.5.0-cdh5.3.6-bin
drwxr-xr-x   6 root root 4096 Dec 18 09:19 apache-maven-3.3.9
drwxr-xr-x.  2 root root 4096 Sep 23  2011 bin
drwxr-xr-x.  2 root root 4096 Sep 23  2011 etc
drwxr-xr-x.  2 root root 4096 Sep 23  2011 games
drwxr-xr-x  10 root root 4096 Dec 17 10:11 hadoop-2.5.0-cdh5.3.6
drwxr-xr-x   8 root root 4096 Dec 17 10:42 hive-0.13.1-cdh5.3.6
drwxr-xr-x.  2 root root 4096 Sep 23  2011 include
drwxr-xr-x   8 uucp  143 4096 Apr 11  2015 jdk1.7.0_79
drwxr-xr-x.  2 root root 4096 Sep 23  2011 lib
drwxr-xr-x.  2 root root 4096 Sep 23  2011 lib64
drwxr-xr-x.  2 root root 4096 Sep 23  2011 libexec
lrwxrwxrwx   1 root root   18 Dec 18 09:19 maven -> apache-maven-3.3.9
drwxr-xr-x.  2 root root 4096 Sep 23  2011 sbin
drwxr-xr-x.  5 root root 4096 Aug 14  2014 share
drwxr-xr-x  24 1106  592 4096 Dec 18 17:36 sqoop2-1.99.4-cdh5.3.6
drwxr-xr-x.  2 root root 4096 Sep 23  2011 src
[root@namenode ~]#
```

图 11-6 解压 JDK

（2）添加进环境变量，如图 11-7 所示。

```
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done

unset i
unset -f pathmunge

export JAVA_HOME=/usr/local/jdk1.7.0_79/
export PATH=$PATH:$JAVA_HOME/bin

export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HIVE_HOME=/usr/local/hive-0.13.1-cdh5.3.6
export PATH=$PATH:$HIVE_HOME/bin

export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
[root@namenode ~]#
```

图 11-7 配置环境变量

（3）加载并验证，配置环境变量之后，执行 source /etc/profile,刷新环境变量，然后执行 java -version 查看 java 的版本。如图 11-8 所示。

```
        fi
    fi
done

unset i
unset -f pathmunge

export JAVA_HOME=/usr/local/jdk1.7.0_79/
export PATH=$PATH:$JAVA_HOME/bin

export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HIVE_HOME=/usr/local/hive-0.13.1-cdh5.3.6
export PATH=$PATH:$HIVE_HOME/bin

export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
[root@namenode ~]# source /etc/profile
[root@namenode ~]# java -version
java version "1.7.0_79"
Java(TM) SE Runtime Environment (build 1.7.0_79-b15)
Java HotSpot(TM) 64-Bit Server VM (build 24.79-b02, mixed mode)
[root@namenode ~]#
```

图 11-8 验证 Java

### 11.5.3 配置 Hadoop

（1）解压 Hadoop 压缩包至/usr/local，执行命令 tar -zxvf/usr/local/hadoop-2.5.0-cdh5.3.6.tar.gz-C/usr/local/，产生 Hadoop 解压包文件，如图 11-9 所示。

```
[root@namenode ~]# ll /usr/local/
total 68
drwxr-xr-x   6 root root 4096 Dec 17 13:42 aegis
drwxr-xr-x   8 1106 4001 4096 Dec 17 13:48 apache-flume-1.5.0-cdh5.3.6-bin
drwxr-xr-x   6 root root 4096 Dec 18 09:19 apache-maven-3.3.9
drwxr-xr-x.  2 root root 4096 Sep 23  2011 bin
drwxr-xr-x.  2 root root 4096 Sep 23  2011 etc
drwxr-xr-x.  2 root root 4096 Sep 23  2011 games
drwxr-xr-x  10 root root 4096 Dec 17 10:11 hadoop-2.5.0-cdh5.3.6
drwxr-xr-x   8 root root 4096 Dec 17 10:42 hive-0.13.1-cdh5.3.6
drwxr-xr-x.  2 root root 4096 Sep 23  2011 include
drwxr-xr-x   8 uucp  143 4096 Apr 11  2015 jdk1.7.0_79
drwxr-xr-x.  2 root root 4096 Sep 23  2011 lib
drwxr-xr-x.  2 root root 4096 Sep 23  2011 lib64
drwxr-xr-x.  2 root root 4096 Sep 23  2011 libexec
lrwxrwxrwx   1 root root   18 Dec 18 09:19 maven -> apache-maven-3.3.9
drwxr-xr-x.  2 root root 4096 Sep 23  2011 sbin
drwxr-xr-x.  5 root root 4096 Aug 14  2014 share
drwxr-xr-x  24 1106  592 4096 Dec 18 17:36 sqoop2-1.99.4-cdh5.3.6
drwxr-xr-x.  2 root root 4096 Sep 23  2011 src
[root@namenode ~]#
```

图 11-9 解压 Hadoop

（2）修改环境变量，解压 Hadoop 压缩包之后，需要修改环境配置变量，执行命令 vim /etc/profile，并在文件中添加如下内容：

```
export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

如图 11-10 所示。

```
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done

unset i
unset -f pathmunge

export JAVA_HOME=/usr/local/jdk1.7.0_79/
export PATH=$PATH:$JAVA_HOME/bin

export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HIVE_HOME=/usr/local/hive-0.13.1-cdh5.3.6
export PATH=$PATH:$HIVE_HOME/bin

export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
"/etc/profile" 90L, 2170C                                    90,1          Bot
```

图 11-10 配置环境变量

（3）验证并查看 Hadoop 版本，如图 11-11 所示。

（4）修改配置文件。

```
[root@namenode ~]# source /etc/profile
[root@namenode ~]# hadoop version
Hadoop 2.5.0-cdh5.3.6
Subversion Unknown -r Unknown
Compiled by root on 2016-12-17T00:18Z
Compiled with protoc 2.5.0
From source with checksum 9c7775296a534f91809cc23d2d15ffcc
This command was run using /usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/common/
hadoop-common-2.5.0-cdh5.3.6.jar
[root@namenode ~]#
```

图 11-11 验证 Hadoop

### core-site.xml

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/core-site.xml 并修改文件内容如下：

```
<configuration>
<property>
<name>hadoop.tmp.dir</name>
<value>/data/tmp</value>
<description>Abase for other temporary directories.</description>
</property>
<property>
<name>fs.defaultFS</name>
<value>hdfs://namenode:9000</value>
</property>
<property>
<name>io.file.buffer.size</name>
<value>4096</value>
</property>
</configuration>
```

### hdfs-site.xml

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/hdfs-site.xml 并修改文件内容如下：

```
<configuration>
<property>
<name>dfs.namenode.secondary.http-address</name>
<value>namenode:50090</value>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:///data/dfs/name</value>
</property>
```

```
<property>
<name>dfs.datanode.data.dir</name>
<value>file:///data/dfs/data</value>
</property>
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
<property>
<name>dfs.webhdfs.enabled</name>
<value>true</value>
</property>
</configuration>
```

yarn-site.xml

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/yarn-site.xml 并修改文件内容如下：

```
<configuration>
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.resourcemanager.hostname</name>
<value>namenode</value>
</property>
<property>
<name>yarn.web-proxy.address</name>
<value>namenode:8041</value>
</property>
</configuration>
```

mapred-site.xml

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/mapred-site.xml 并修改文件内容如下：

```
<configuration>
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
<property>
```

```
<name>mapreduce.jobtracker.http.address</name>
<value>namenode:50030</value>
</property>
<property>
<name>mapreduce.jobhistory.address</name>
<value>namenode:10020</value>
</property>
<property>
<name>mapreduce.jobhistory.webapp.address</name>
<value>namenode:19888</value>
</property>
</configuration>
```

### hadoop-env.sh

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/hadoop-env.sh 并修改文件内容如下：

在文件开始加入

```
export JAVA_HOME=/usr/local/jdk1.7.0_79/
```

### yarn-env.sh

执行命令 vim /usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/yarn-env.sh 并修改文件内容如下：

在文件开始加入

```
export JAVA_HOME=/usr/local/jdk1.7.0_79/
```

（5）格式化文件系统。在命令终端执行命令 hdfs namenode -format 实现对 HDFS 系统的格式化，如图 11-12 所示。

```
[root@namenode ~]# hdfs namenode -format
```

图 11-12 格式化文件系统

（6）启动 HDFS，在命令终端执行命令 start-dfs.sh 启动 HDFS 服务进程，如图 11-13 所示。

```
[root@namenode ~]# start-dfs.sh
Starting namenodes on [namenode]
namenode: starting namenode, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/logs/ha
doop-root-namenode-namenode.out
localhost: starting datanode, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/logs/h
adoop-root-datanode-namenode.out
Starting secondary namenodes [namenode]
namenode: starting secondarynamenode, logging to /usr/local/hadoop-2.5.0-cdh5.3.
6/logs/hadoop-root-secondarynamenode-namenode.out
[root@namenode ~]# jps
2310 NameNode
2677 Jps
2402 DataNode
2573 SecondaryNameNode
[root@namenode ~]#
```

图 11-13 启动 HDFS

（7）查看 Web 页面，启动 HDFS 服务之后，在浏览器地址栏中，输入 http://139.129.226.141:50070，查看 HDFS 系统页面。如图 11-14 所示。

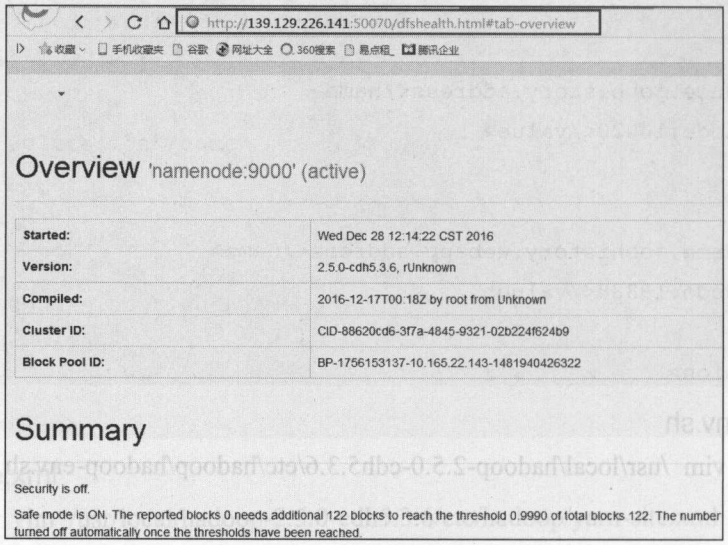



图 11-14　页面查看 HDFS

（8）修改 Yarn 启动脚本。

增加 proxyserver 和 historyserver 的启动项。

vim /usr/local/hadoop-2.5.0-cdh5.3.6/sbin/start-yarn.sh

取消注释："$bin"/yarn-daemon.sh --config $YARN_CONF_DIR　stop proxyserver。

添加 "$bin"/mr-jobhistory-daemon.sh --config $YARN_CONF_DIR start historyserver

vim /usr/local/hadoop-2.5.0-cdh5.3.6/sbin/stop-yarn.sh。

添加 "$bin"/mr-jobhistory-daemon.sh --config $YARN_CONF_DIR stop historyserver。

（9）启动 Yarn ，在任意目录下，执行命令 start-yarn.sh，启动 Yarn 所有服务进程。如图 11-15 所示。

```
[root@namenode usr]# start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/logs/yarn-
root-resourcemanager-namenode.out
localhost: starting nodemanager, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/log
s/yarn-root-nodemanager-namenode.out
starting proxyserver, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/logs/yarn-root
-proxyserver-namenode.out
starting historyserver, logging to /usr/local/hadoop-2.5.0-cdh5.3.6/logs/mapred-
root-historyserver-namenode.out
[root@namenode usr]#
```

图 11-15　启动 Yarn

（10）测试 wordcount，在任意目录下，执行命令 hadoop jar/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.5.0-cdh5.3.6.jar wordcount /test/wc.txt /output/1。如图 11-16 所示。

```
3344 NodeManager
[root@namenode usr]# hadoop jar /usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/ma
preduce/hadoop-mapreduce-examples-2.5.0-cdh5.3.6.jar wordcount /test/wc.txt /out
put/1
16/12/28 13:17:48 INFO client.RMProxy: Connecting to ResourceManager at namenode
/139.129.226.141:8032
16/12/28 13:17:49 INFO input.FileInputFormat: Total input paths to process : 1
16/12/28 13:17:49 INFO mapreduce.JobSubmitter: number of splits:1
16/12/28 13:17:50 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_14
82902074524_0001
16/12/28 13:17:50 INFO impl.YarnClientImpl: Submitted application application_14
82902074524_0001
16/12/28 13:17:50 INFO mapreduce.Job: The url to track the job: http://namenode:
8041/proxy/application_1482902074524_0001/
16/12/28 13:17:50 INFO mapreduce.Job: Running job: job_1482902074524_0001
16/12/28 13:18:04 INFO mapreduce.Job: Job job_1482902074524_0001 running in uber
 mode : false
16/12/28 13:18:04 INFO mapreduce.Job:  map 0% reduce 0%
16/12/28 13:18:11 INFO mapreduce.Job:  map 100% reduce 0%
16/12/28 13:18:20 INFO mapreduce.Job:  map 100% reduce 100%
16/12/28 13:18:20 INFO mapreduce.Job: Job job_1482902074524_0001 completed succe
ssfully
16/12/28 13:18:20 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=887
                FILE: Number of bytes written=209781
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
```

图 11-16　测试 wordcount

（11）查看 wordcount 结果，上一步执行完之后，执行命令 hadoop fs -ls /output/1 和 hadoop fs -cat /output/1/part-r-00000，分别查看生成的目录和生成的结果。如图 11-17 所示。

```
                Bytes Written=613
[root@namenode usr]# clear
[root@namenode usr]# hadoop fs -ls /output/1
Found 2 items
-rw-r--r--   2 root supergroup          0 2016-12-28 13:18 /output/1/_SUCCESS
-rw-r--r--   2 root supergroup        613 2016-12-28 13:18 /output/1/part-r-0000
0
[root@namenode usr]# hadoop fs -cat /output/1/part-r-00000
Apache  1
Hadoop  1
It      1
Rather  1
The     1
a       3
across  1
allows  1
and     2
application     1
at      1
be      1
cluster 1
clusters        1
computation     1
computers       1
computers,      1
data    1
deliver 1
delivering      1
```

图 11-17　查看目录

### 11.5.4 配置 Hive

（1）执行 yum install -y mysql-server mysql mysql-devel 安装 MySQL。如图 11-18 所示。

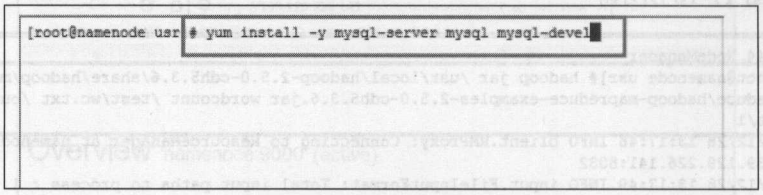



图 11-18 安装 MySQL

（2）启动 MySQL，在命令终端，执行命令 service mysqld start，启动 MySQL 服务。如图 11-19 所示。

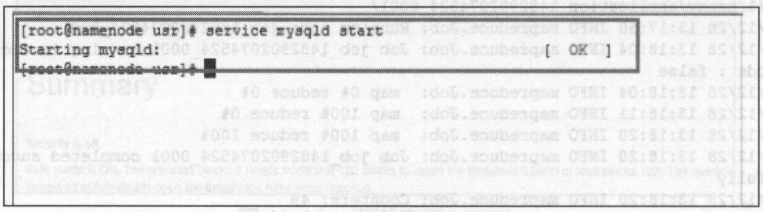



图 11-19 启动 MySQL

（3）修改 root 密码，在任意目录下，执行命令 mysqladmin -u root -ppasswordroot，进行密码修改，在提示的 enter password：之后输入新的密码。如图 11-20 所示。

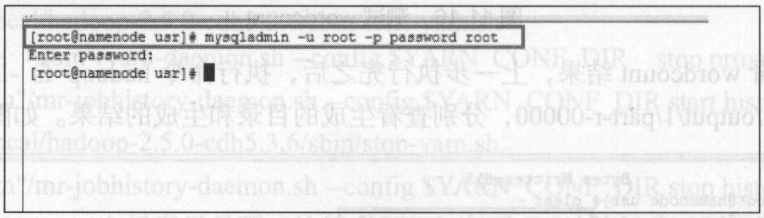



图 11-20 修改密码

（4）登录并操作验证，在任意目录下，执行命令 mysql -uroot -proot，进入到 mysql 环境中。如图 11-21 所示。

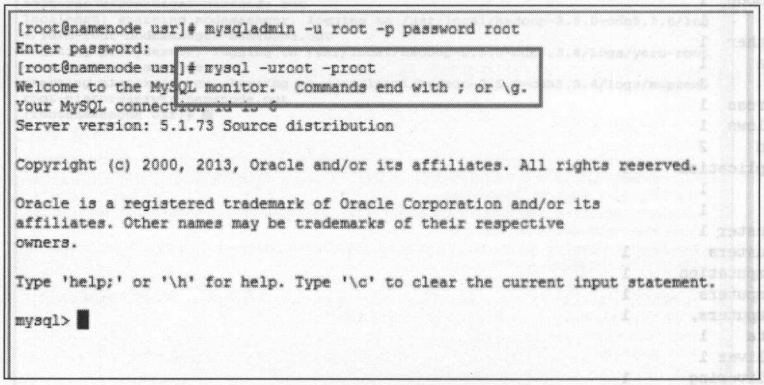



图 11-21 登录 MySQL

（5）在 MySQL 命令行下，进行相关操作：查看数据库、查看表。如图 11-22 所示。

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| hive               |
| mysql              |
| ota                |
| test               |
+--------------------+
5 rows in set (0.01 sec)

mysql> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| user           |
+----------------+
1 row in set (0.00 sec)

mysql>
```

图 11-22 MySQL 操作

（6）在 MySQL 命令行下，进行相关操作：创建表、插入数据、查询。如图 11-23 所示。

```
mysql> drop table user;
Query OK, 0 rows affected (0.00 sec)

mysql> create table user(id int,name varchar(20));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into user values(1,'aa');
Query OK, 1 row affected (0.00 sec)

mysql> select * from user;
+------+------+
| id   | name |
+------+------+
|    1 | aa   |
+------+------+
1 row in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
```

图 11-23 创建表

（7）解压 Hive，执行命令 tar -zxvf /soft/hive-0.13.1-cdh5.3.6.tar.gz -C /usr/local/，实现 hive 压缩包解压。如图 11-24 所示。

```
[root@namenode usr]# tar -zxvf /soft/hive-0.13.1-cdh5.3.6.tar.gz -C /usr/local/
```

图 11-24 解压 Hive

（8）拷贝 MySQL 驱动，执行命令 cp /soft/mysql-connector-java-6.0.5.jar /usr/local/hive-0.13.1-cdh5.3.6/lib/，把 MySQL 的 jar 包拷贝到 hive 的目录下。如图 11-25 所示。

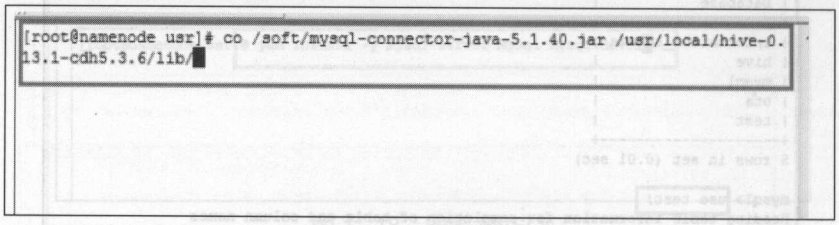



图 11-25　拷贝 MySQL 驱动

（9）修改配置文件，在命令终端执行命令 vim /usr/local/hive-0.13.1-cdh5.3.6/conf/hive-site.xml 并修改文件的内容如下：

```
<configuration>
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:mysql://localhost:3306/hive?createDatabaseIfNotExist=true&amp;
autoReconnect=true&amp;characterEncoding=UTF-8</value>
</property>
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>root</value>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>root</value>
</property>
<property>
<name>datanucleus.autoCreateSchema</name>
<value>true</value>
</property>
<property>
<name>datanucleus.fixedDatastore</name>
<value>false</value>
</property>
<property>
<name>hive.metastore.uris</name>
```

```
<value>thrift://namenode:9083</value>
</property>
</configuration>
```

（10）修改环境变量，执行命令 vim /etc/profile 并编辑文件中的内容，然后执行命令 source /etc/profile，进行刷新环境变量。如图 11-26 所示。

```
    umask 022
fi

for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done

unset i
unset -f pathmunge

export JAVA_HOME=/usr/local/jdk1.7.0_79/
export PATH=$PATH:$JAVA_HOME/bin

export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HIVE_HOME=/usr/local/hive-0.13.1-cdh5.3.6
export PATH=$PATH:$HIVE_HOME/bin

export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
"/etc/profile" 90L, 2170C                                    90,1          Bot
```

图 11-26 修改环境变量

（11）在命令终端的任意目录下，执行命令 hadoop fs -mkdir /customer，在 HDFS 系统中创建一个 customer 目录。如图 11-27 所示。

```
[root@namenode sample]# hadoop fs -mkdir /customer
[root@namenode sample]# hadoop fs -ls /
Found 9 items
drwxr-xr-x   - root supergroup          0 2016-12-18 12:45 /Trash
drwxr-xr-x   - root supergroup          0 2016-12-28 14:02 /customer
drwxr-xr-x   - root supergroup          0 2016-12-18 12:56 /format
drwxr-xr-x   - root supergroup          0 2016-12-28 13:18 /output
drwxr-xr-x   - root supergroup          0 2016-12-18 13:05 /result
drwxr-xr-x   - root supergroup          0 2016-12-18 12:47 /source
drwxr-xr-x   - root supergroup          0 2016-12-28 12:30 /test
drwx------   - root supergroup          0 2016-12-18 13:02 /tmp
drwxr-xr-x   - root supergroup          0 2016-12-18 13:02 /user
[root@namenode sample]#
```

图 11-27 创建 customer 目录

（12）在命令终端的任意目录下，执行命令 hadoop fs -put /sample/1.txt /customer,把本地的文件 1.txt 上传到 HDFS 系统中，如图 11-28 所示。

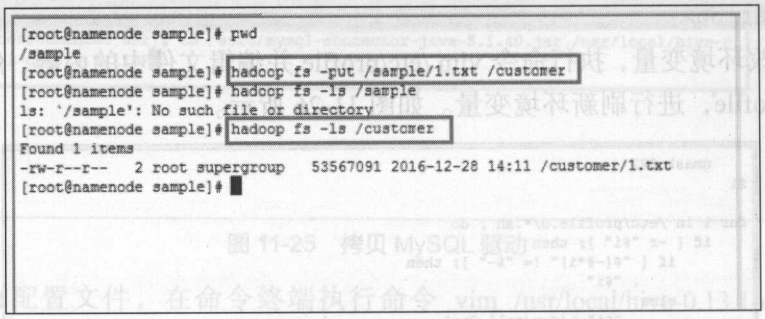



图 11-28　上传文件

（13）启动 Hive 服务，分别在不同的命令窗口启动服务。如图 11-29 所示。

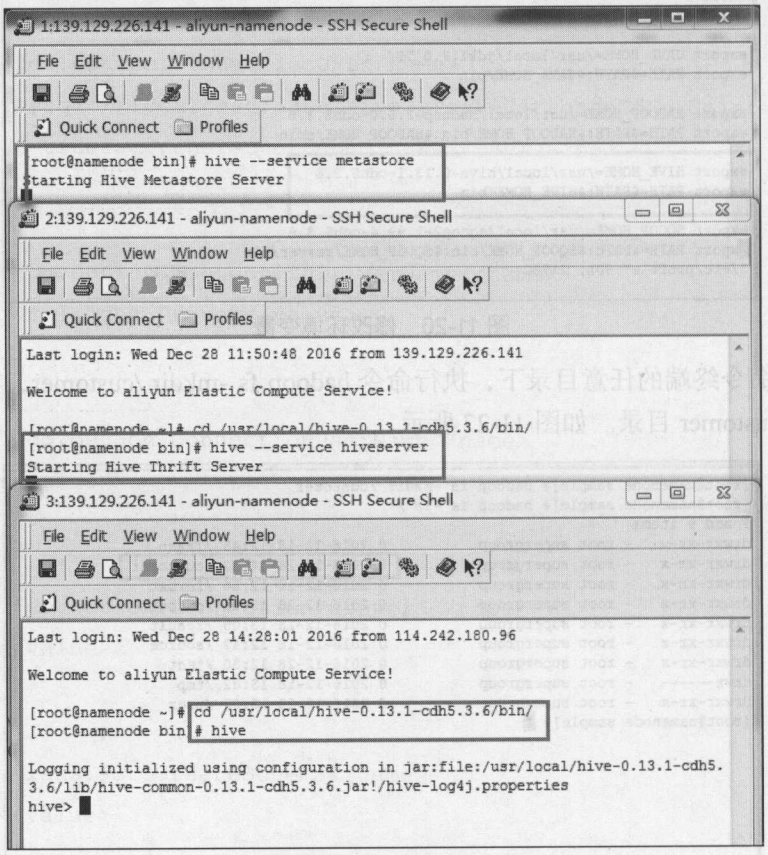



图 11-29　启动 Hive

（14）在 Hive 的 Shell 命令，进行如下命令操作：

```
hive> use default;
hive>
CREATE EXTERNAL TABLE'customer' ('id' int, 'name' string)  ROW FORMAT DELIMITED
> FIELDS TERMINATED BY '\t'
```

```
> LINES TERMINATED BY '\n'
> LOCATION
> 'hdfs://namenode:9000/customer';
```

如图 11-30 所示。

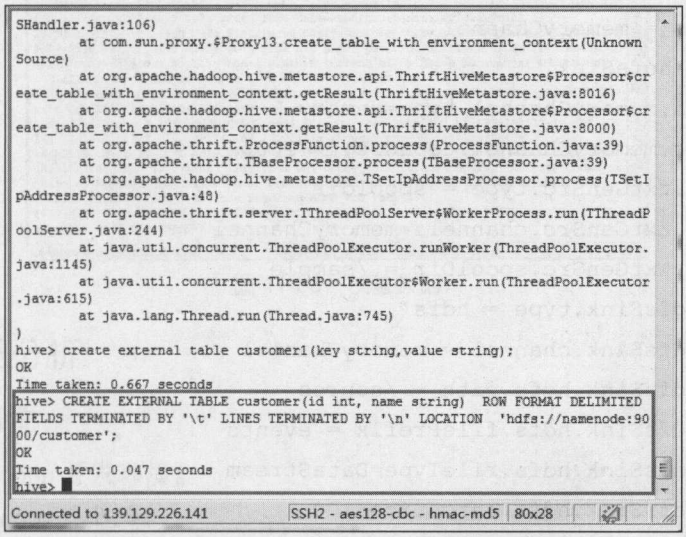



图 11-30 创建表 customer

说明

创建表时，如果出现错误“Specified key was too long; max key length is 767 bytes”，则可以在 mysql 命令行中，执行 mysql > alter database hive character set latin1;即可解决。

（15）在 hive 的 shell 命令，进行如下命令操作：select * from customer;查询数据。

```
OK
1   a
2   b
3   c
4   d
```

## 11.6 数据收集

### 11.6.1 解压 Flume

解压 Flume 压缩包至/usr/local，执行命令 tar -zxvf/soft/flume-ng-1.5.0-cdh5.3.6.tar.gz -C /usr/local/，产生 Flume 解压包文件，如图 11-31 所示。

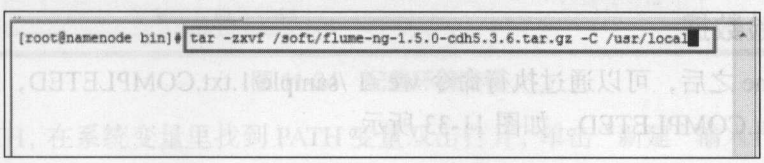



图 11-31 解压 Flume

### 11.6.2 修改配置文件

在命令终端，执行命令：

```
vim /usr/local/apache-flume-1.5.0-cdh5.3.6-bin/conf/agent.properties
agent.sources = txtGenSrc
agent.channels = memoryChannel
agent.sinks = hdfsSink
agent.channels.memoryChannel.type=memory
agent.channels.memoryChannel.capacity = 1000
agent.sources.txtGenSrc.type = spooldir
agent.sources.txtGenSrc.channels=memoryChannel
agent.sources.txtGenSrc.spoolDir = /sample
agent.sinks.hdfsSink.type = hdfs
agent.sinks.hdfsSink.channel = memoryChannel
agent.sinks.hdfsSink.hdfs.path = /source
agent.sinks.hdfsSink.hdfs.filePrefix = events
agent.sinks.hdfsSink.hdfs.fileType=DataStream
agent.sinks.hdfsSink.hdfs.batchSize=100
agent.sinks.hdfsSink.hdfs.round = true
agent.sinks.hdfsSink.hdfs.rollSize=20971520
agent.sinks.hdfsSink.hdfs.rollInterval = 30
agent.sinks.hdfsSink.hdfs.rollCount=0
```

### 11.6.3 启动 Flume

在命令终端，执行命令：

```
/usr/local/apache-flume-1.5.0-cdh5.3.6-bin/bin/flume-ng agent -n agent -f
/usr/local/apache-flume-1.5.0-cdh5.3.6-bin/conf/agent.properties
```

启动 Flume。如图 11-32 所示。

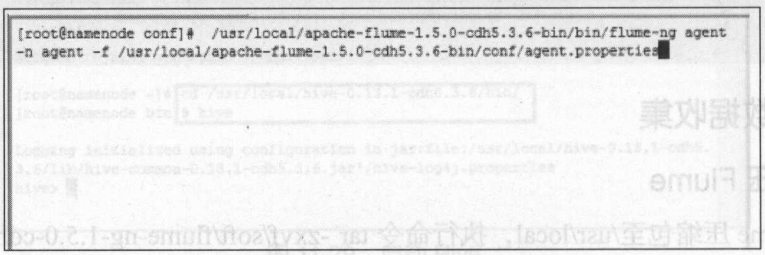



图 11-32 启动 Flume

### 11.6.4 校验数据

启动 flume 之后，可以通过执行命令 wc -l /sample/1.txt.COMPLETED，查看产生的文件/sample/1.txt.COMPLETED。如图 11-33 所示。

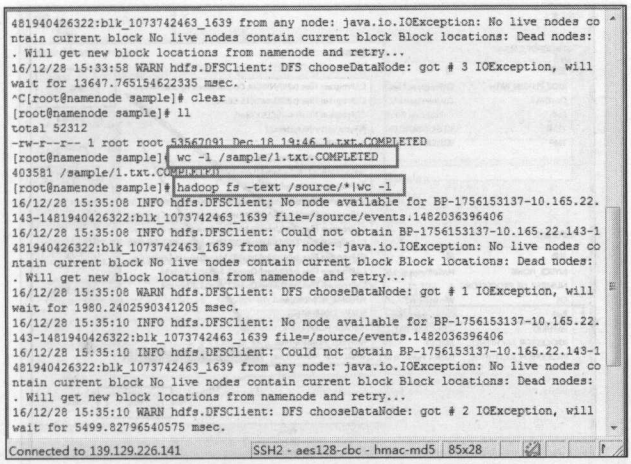



图 11-33 校验数据

## 11.7 数据分析

### 11.7.1 数据清洗

（1）配置 Hadoop 开发环境。

（2）配置 Hadoop 本地库。

新建目录 E:\hadoop-2.5.0-cdh5.3.6。

解压 HadoopBin.zip 至 E:\hadoop-2.5.0-cdh5.3.6。

（3）配置环境变量 HADOOP_HOME。

并将 HADOOP_HOME/bin 加入 PATH 中。右击计算机选择“属性”→“高级系统设置”，找到“高级”选择里面的“环境变量”，在系统变量里“新建”“HADOOP_HOME”变量，如图 11-34 所示。

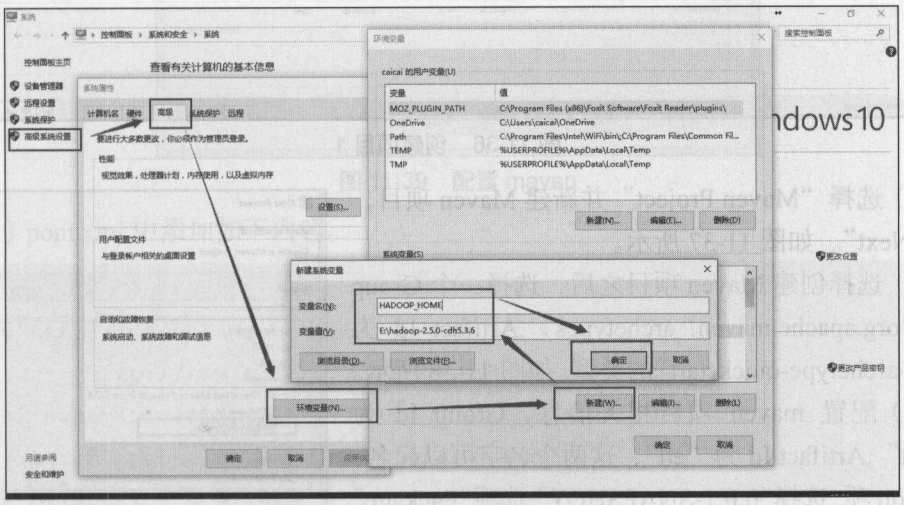



图 11-34 配置环境变量

配置 PATH，在系统变量里找到 PATH 变量双击打开，单击“新建”输入“%HADOOP_HOME%\bin”，单击“确定”完成操作，如图 11-35 所示。

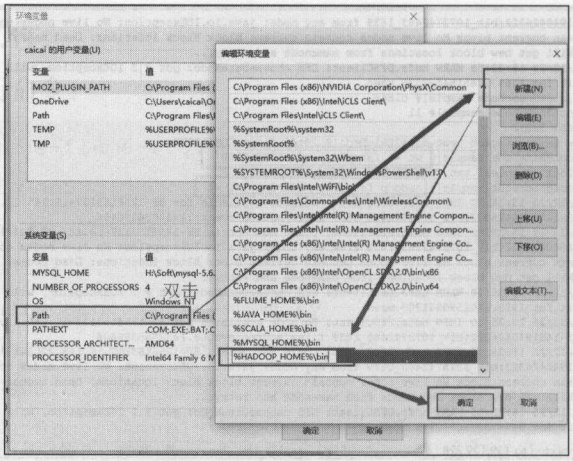



图 11-35　配置 PATH

（4）打开 Eclipse，单击右键“Project Explorer”项目列表空间，选择“New”→“Project”新建 Maven 项目。如图 11-36 所示。

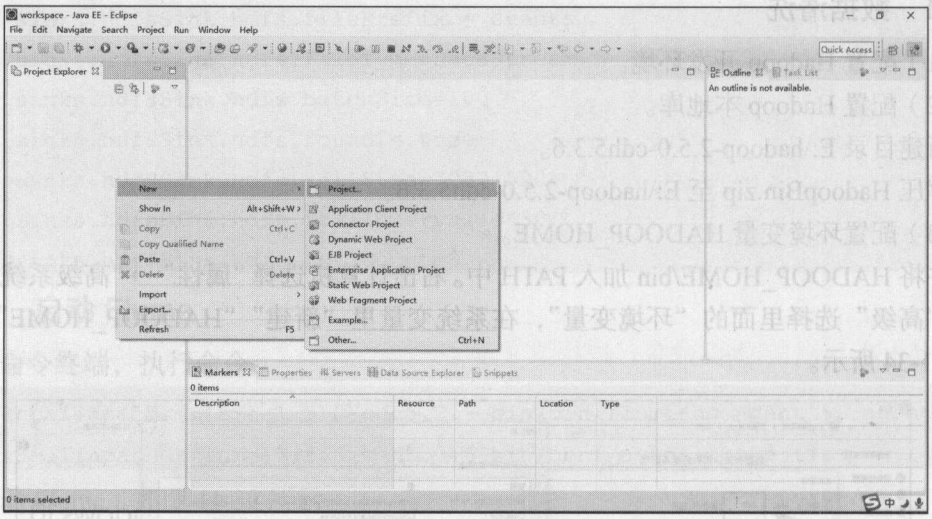



图 11-36　创建项目 1

（5）选择“Maven Project”并新建 Maven 项目，选择“Next”。如图 11-37 所示。

（6）选择创建 Maven 项目之后，选择一个 Group Id 为“org.apache.maven. archetypes”,Artifact Id 为“maven-archetype-quickstart”的类型。如图 11-38 所示。

（7）配置 maven 项目相关信息，Group Id 为“org.test”,Artifact Id 为“etl”，这两个名字可以起名，在 Version 项，选择“0.0.1-SNAPSHOT”,选项“Package”是自动生成的。如图 11-39 所示。

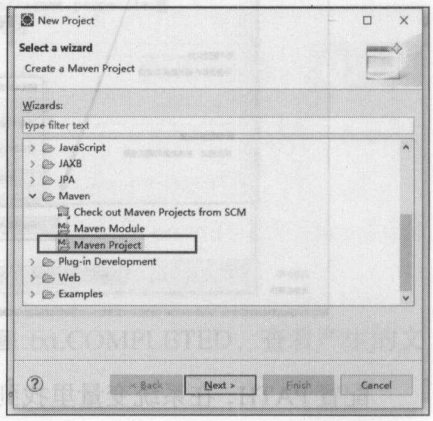



图 11-37　创建项目 2

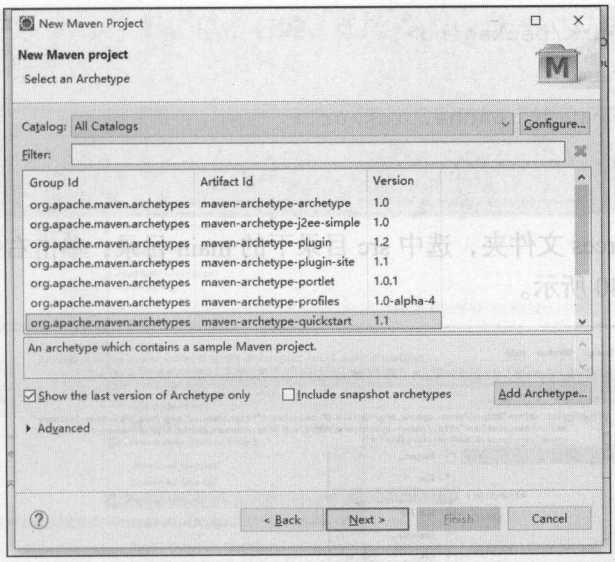



图 11-38 选择 archetype 类型

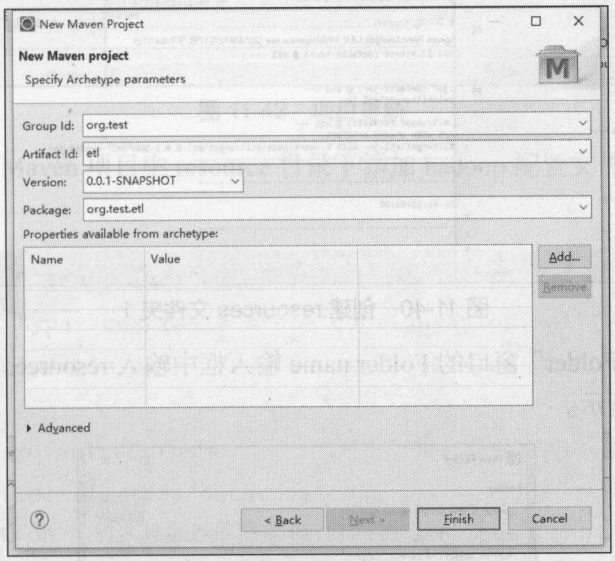



图 11-39 配置 maven

（8）pom.xml 中添加如下内容。

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.test</groupId>
    <artifactId>etl</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

```
    <packaging>jar</packaging>
    <name>etl</name>
    <url>http://maven.apache.org</url>
    ..........................
</project>
```

（9）创建 resources 文件夹，选中 src 目录下的 main 目录，单击右键，选择“New”→“Folder”。如图 11-40 所示。

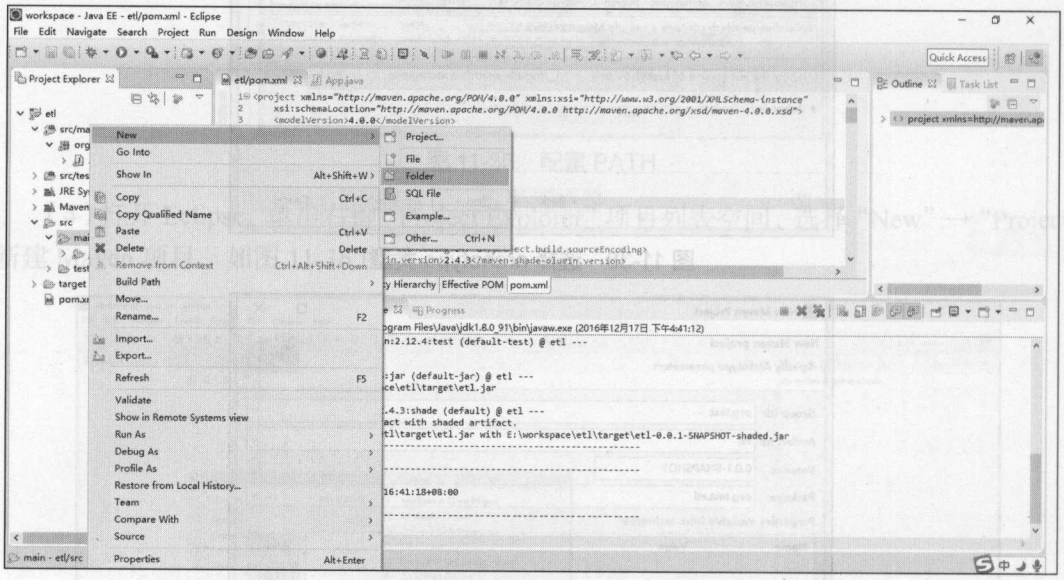



图 11-40　创建 resources 文件夹 1

（10）在“New Folder”窗口的 Folder name 输入框中输入 resources，然后单击“Finish”按钮，如图 11-41 所示。

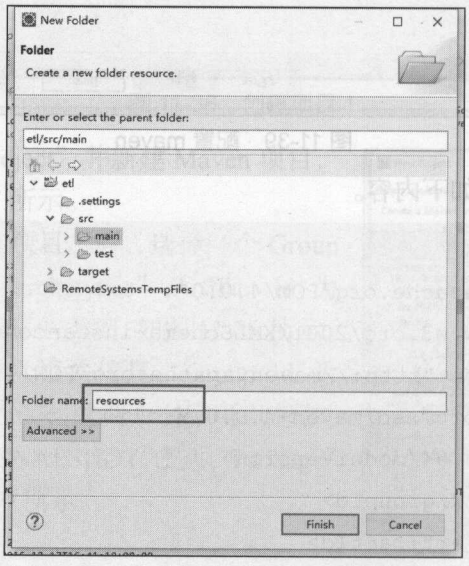



图 11-41　创建 resources 文件夹 2

（11）选中创建的 Maven 项目，单击右键，选择“Maven”→“Update Project”进行项目更新，如图 11-42 所示。

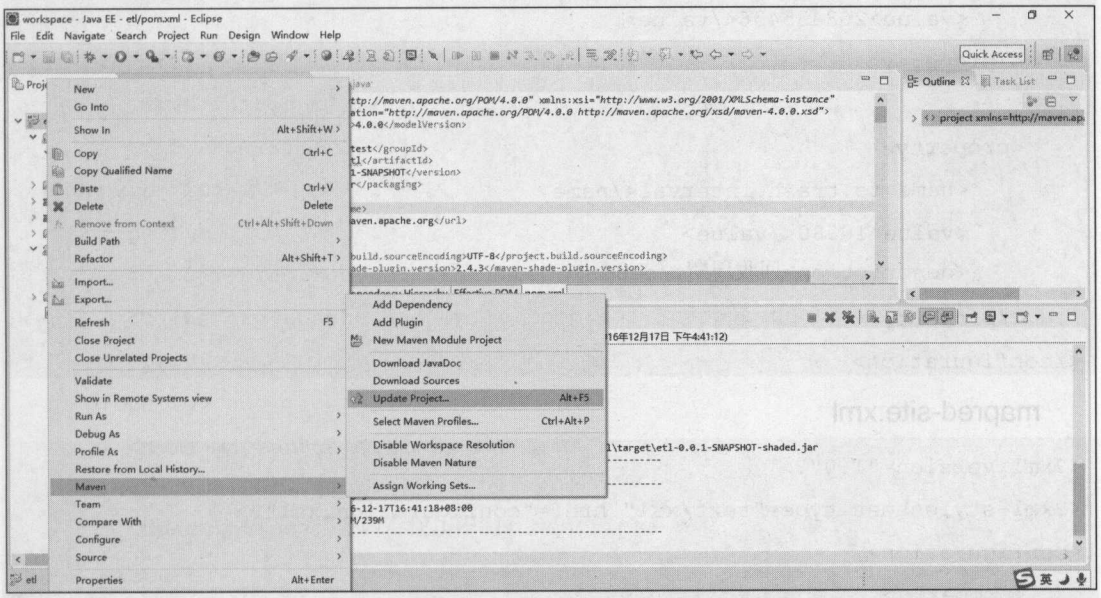



图 11-42 项目更新

（12）在创建的 Maven 项目中 resource 目录下添加 hadoop 配置文件。

core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
    <property>
        <name>fs.defaultFS</name>
        <value>file:///e:/dfs/data</value>
        <description>缺省文件服务的协议和 NS 逻辑名称，和 hdfs-site 里的对应此配置替代了 1.0 里的 fs.default.name</description>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:///e:/dfs/tmp</value>
    </property>
</configuration>
```

hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
```

```
    <property>
        <name>dfs.blocksize</name>
        <value>268435456</value>
    </property>

    <property>
        <name>fs.trash.interval</name>
        <value>10080</value>
        <description>回收周期</description>
    </property>
</configuration>
```

mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>local</value>
    </property>
    <property>
        <name>mapreduce.application.classpath</name>
        <value>

    $HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*,$HADOOP_MAPRED_HOME/share/hadoo
p/mapreduce/lib/*:$HADOOP_CONF_DIR,$HADOOP_COMMON_HOME/share/hadoop/common/*,
    $HADOOP_COMMON_HOME/share/hadoop/common/lib/*,$HADOOP_COMMON_HOME/lib/*,
    $HADOOP_HDFS_HOME/share/hadoop/hdfs/*,$HADOOP_HDFS_HOME/share/hadoop/h
dfs/lib/*,
    $HADOOP_YARN_HOME/share/hadoop/yarn/*,$HADOOP_YARN_HOME/share/hadoop/y
arn/lib/*,$HADOOP_HOME/lib/native/Linux-amd64-64/*,$HADOOP_HOME/lib/*,
    $HADOOP_YARN_HOME/share/hadoop/mapreduce/*,$HADOOP_YARN_HOME/share/had
oop/mapreduce/lib/*,$HADOOP_HOME/share/hadoop/common/lib/hadoop-lzo.jar,$H
ADOOP_HOME/share/hadoop/common,$HADOOP_HOME/lib/
        </value>
    </property>
    <property>
        <name>mapreduce.local.dir</name>
```

```
        <value>file:///e:/dfs/mapred</value>
    </property>

    <property>
        <name>mapreduce.map.output.compress</name>
        <value>false</value>
    </property>
    <property>
        <name>mapred.map.output.compress.codec</name>
        <value>org.apache.hadoop.io.compress.SnappyCodec</value>
    </property>
    <property>
        <name>mapreduce.task.io.sort.mb</name>
        <value>100</value>
        <description>排序内存使用限制</description>
    </property>
</configuration>
```

log4j.properties

```
log4j.rootLogger=INFO,stdout,R
# stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] %c{2}:%L %m%n
log4j.appender.stdout.Encoding=UTF-8

# rolling log file
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.maxFileSize=1GB
log4j.appender.R.maxBackupIndex=10
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %c (line %L)
%m%n
log4j.appender.R.File=logs/server.log
log4j.appender.R.Encoding=UTF-8
```

（13）基于以上的操作，最终构成项目结构，如图11-43所示。

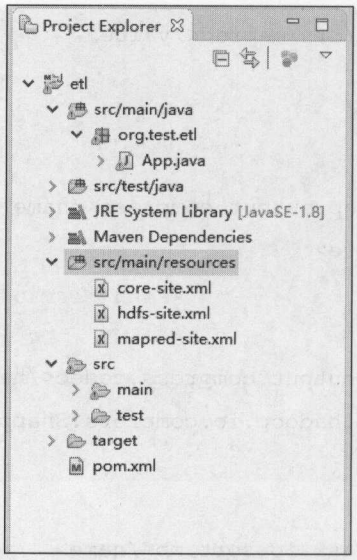



图 11-43　项目结构

### 11.7.2　ETL 编程

（1）创建 etl 项目之后，编写 MapReduce 主程序 org.test.etl.App。

```
package org.test.etl;
public class App extends Configured implements Tool {
    private static Logger logger = LoggerFactory.getLogger(App.class);
    public static class EtlMapper extends Mapper<LongWritable, Text, Text,
NullWritable> {
        private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        private DateFormat dateTimeFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            String line = value.toString();
            String[] arr = line.split("\t");
            if (arr.length != 9) {
            logger.warn("length  not  match:len:{},  line:{}",  arr.length,
line);
                return;
            }
            String orderid = arr[0];
            String provincename = arr[4];
            if (StringUtils.isBlank(orderid) || StringUtils.isBlank(provincename))
{
                logger.error("orderid  or  provincename  can't  be  empty!:
```

```
orderid:{}, provincename:{}", orderid, provincename);
                return;
            }
            String hotelid = arr[1];
            String provinceid = arr[3];
            if(!NumberUtils.isNumber(hotelid)
|| !NumberUtils.isNumber(provinceid)) {
                logger.error("not a number: hotelid: {}, provinceid: {}",
hotelid, provinceid);
                return;
            }
            String arrivaldateString = arr[5];
            try {
                Date arrivaldate = dateTimeFormat.parse(arrivaldateString);
                arr[5] = dateFormat.format(arrivaldate);
            } catch (ParseException e) {
                logger.error("not date:{}, {}", arrivaldateString, line);
                return;
            }
            String departuredateString = arr[6];
            try {
                Date departuredate = dateTimeFormat.parse(departuredateString);
                arr[6] = dateFormat.format(departuredate);
            } catch (ParseException e) {
                logger.error("not date:{}, {}", arrivaldateString, line);
                return;
            }
            context.write(new Text(StringUtils.join(arr, "\t")), NullWritable.
get());
        }
    }
    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            logger.error("args length not match:{}", StringUtils.join(args,
","));
            return -1;
        }
        Configuration conf = getConf();
```

```
        String input = args[0];
        String output = args[1];
        FileSystem fs = FileSystem.get(conf);
        fs.delete(new Path(output), true);
        Job job = Job.getInstance(conf, "OrderEtl");
        job.setJarByClass(App.class);
        job.setMapperClass(EtlMapper.class);
        job.setNumReduceTasks(0);
        // 指定输入
        FileInputFormat.setInputPaths(job, new Path(input));
        job.setInputFormatClass(TextInputFormat.class);
        // 输出类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(NullWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);
        // 输出格式
        FileOutputFormat.setOutputPath(job, new Path(output));
        job.setOutputFormatClass(TextOutputFormat.class);
        // 提交任务
        boolean success = job.waitForCompletion(true);
        if (success) {
            System.out.println("Success");
            return 0;
        } else {
            System.out.println("Failed");
            return -1;
        }
    }
    public static void main(String[] args) throws Exception {
        int ret = 0;
        try {
            ret = ToolRunner.run(new Configuration(), new App(), args);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(ret);
    }
}
```

（2）测试 MapReduce 程序。

准备输入数据，在命令终端，执行命令 head -10 /sample/1.txt >order-10.txt，并把数据放入目录 E:\dfs\data\source\，然后准备运行程序，如图 11-44 所示。

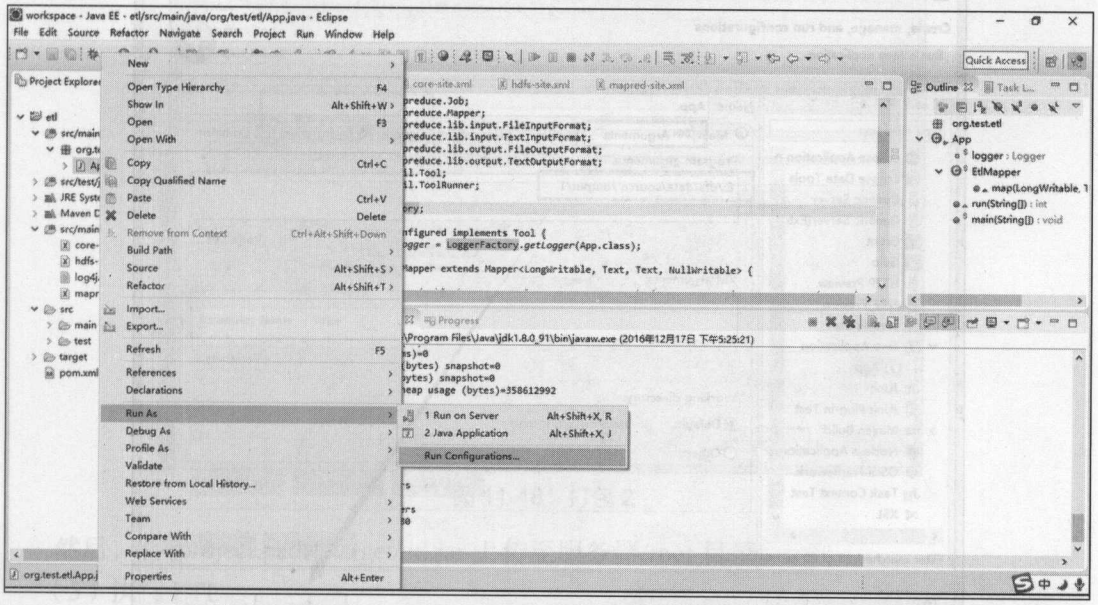



图 11-44 输入数据

单击右键类“App.java”，选择“Run As”→“Run Configuration”，进入项目运行配置界面，如图 11-45 所示。

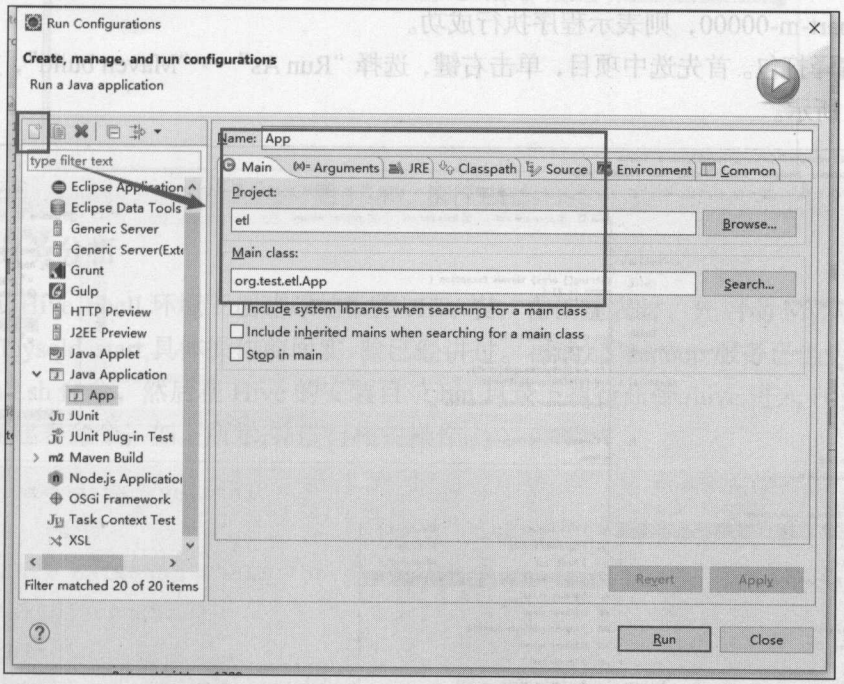



图 11-45 创建运行程序

在“Run Configurations”界面，选择标签项“Arguments”并指定参数列表 E:/dfs/data/source/output/1，注意两个参数中间是空格。如图 11-46 所示。

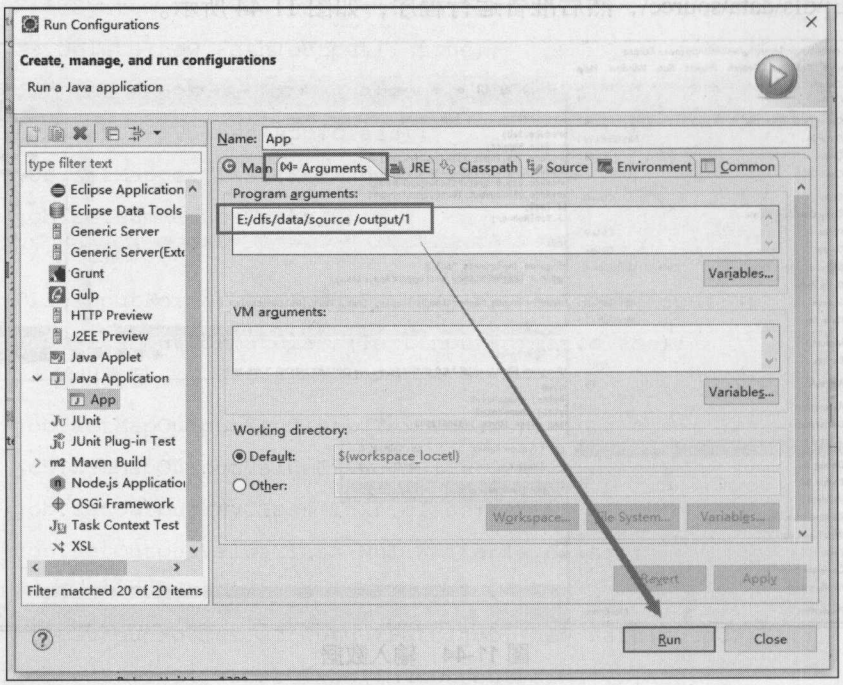



图 11-46　指定参数列表

（3）程序运行完之后，可以到目录 E:\output\1\part-m-00000 中，查看是否存在，如果存在文件 part-m-00000，则表示程序执行成功。

（4）编译打包。首先选中项目，单击右键，选择“Run As”→“Maven build”，如图 11-47 和图 11-48 所示。

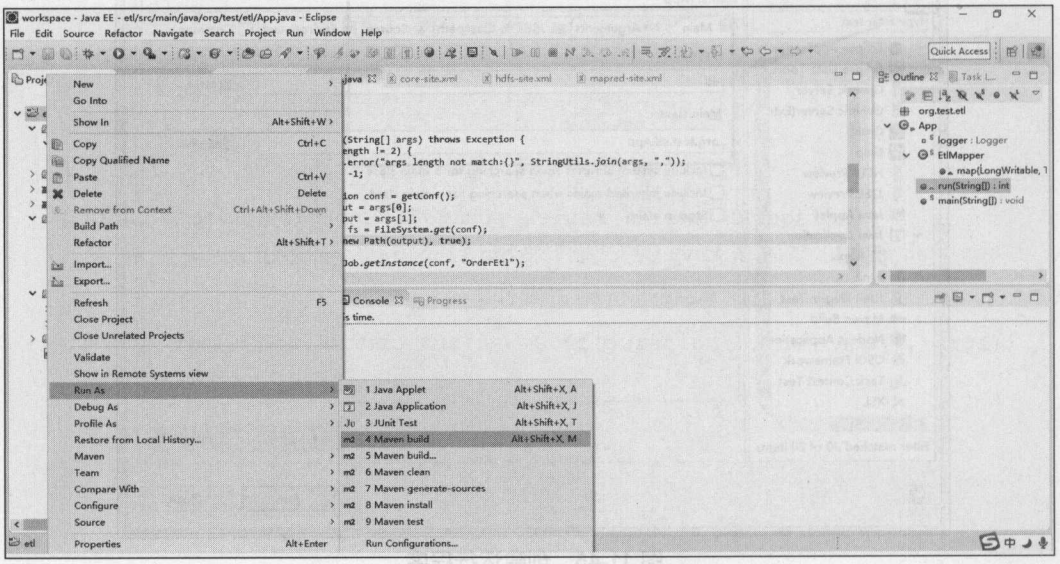



图 11-47　打包 1

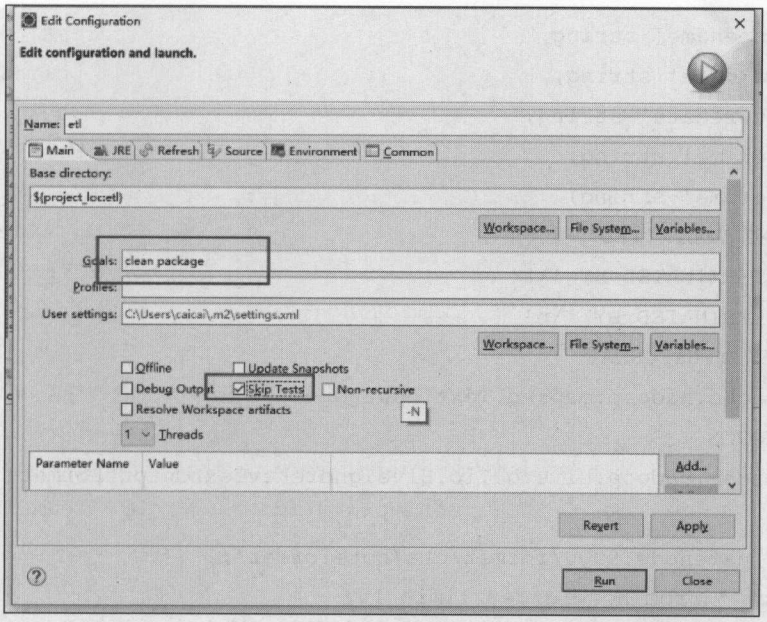



图 11-48 打包 2

然后，将{project.path}/target/etl.jar 上传至服务器/root 目录。

（5）执行 ETL。

在命令行任意目录下执行命令 hadoop jar /root/etl.jar org.test.etl.App /source /format/ota/hotelorder。如图 11-49 所示。

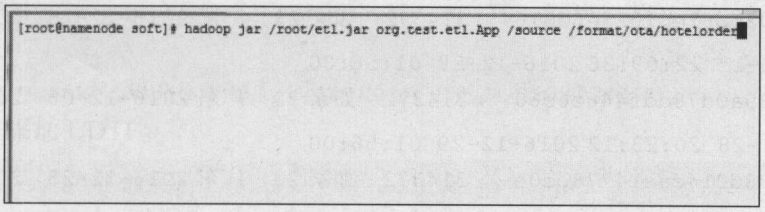



图 11-49 执行 ETL

### 11.7.3 业务分析

（1）在 Hive Shell 环境下创建原始数据 Hive 表，在创建表时，先启动 MySQL 执行命令 service mysqld start,具体操作前面实 验已经讲过。再确定 Hadoop 服务已经启动，否则执行 start-all.sh 命令，然后在 Hive 的安装目录 bin 目录下执行命令./hive 进入 Hive Shell 环境，执行创建表命令，如下所示,并进行相关操作。

```
hive> create database ota;
hive> use ota;
hive> CREATE EXTERNAL TABLE 'hotelorder' (
>   ' orderid' string,
>   ' hotelid' int,
>   ' hotelname' string,
>   ' provinceid' int,
```

```
>   ' provincename' string,
>   ' arrivaldate' string,
>   ' departuredate' string,
>   ' createtime' string,
>   ' updatetime' string)
> ROW FORMAT DELIMITED
>   FIELDS TERMINATED BY '\t'
>   LINES TERMINATED BY '\n'
> STORED AS INPUTFORMAT
>   'org.apache.hadoop.mapred.TextInputFormat'
> OUTPUTFORMAT
>   'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
> LOCATION
>   'hdfs://namenode:9000/format/ota/hotelorder';
hive> select * from hotelorder limit 10;
b628d5ece2a6f41fe676d1116288    314372  如家 23  广东 2016-12-03  2016-12-05
    2016-12-01 23:23:16 2016-12-29 01:56:00
6634c34a6054a7ed6583fedf6a02    314372  如家 23  广东 2016-02-06  2016-02-08
    2016-12-27 11:30:36 2016-12-29 01:56:00
f5b870877d010ea2a5f0802f244c    314372  如家 23  广东 2016-12-10  2016-12-11
    2016-11-29 22:55:48 2016-12-29 01:56:00
960e7e641702bd62cb1027e6d076    314372  如家 23  广东 2016-12-15  2016-12-17
    2016-12-13 22:09:36 2016-12-29 01:56:00
710c26ca0789a0d78ddf446c6c60    314372  如家 23  广东 2016-12-06  2016-12-07
    2016-11-28 20:23:12 2016-12-29 01:56:00
8561fae2ec98d014caa14778cc0e    314372  如家 23  广东 2016-12-25  2016-12-26
    2016-12-17 21:09:52 2016-12-29 01:56:00
379d43bdc501dc21ce93e707059d    314372  如家 23  广东 2016-12-25  2016-12-26
    2016-12-24 17:19:36 2016-12-29 01:56:00
abab793f90d5cbcf4fd72170560e    314372  如家 23  广东 2016-12-03  2016-12-05
    2016-11-30 01:16:59 2016-12-29 01:56:00
7d20bdb7b614ade7e6f3765a6e42    314372  如家 23  广东 2016-12-09  2016-12-10
    2016-12-08 13:06:08 2016-12-29 01:56:00
74c674390684f2f5f274455a52da    314372  如家 23  广东 2016-12-26  2016-12-28
    2016-12-17 18:11:56 2016-12-29 01:56:00
Time taken: 0.519 seconds, Fetched: 10 row(s)
```

（2）创建分析结果表，在 Hive 的安装目录 bin 目录下执行命令./hive 进入 Hive Shell 环境，执行创建表命令，如下所示。

```
CREATE EXTERNAL TABLE 'order_num_bydate'(
  'hotelid' int,
```

```
  'hotelname' string,
  'indate' string,
  'ordernum' int)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LINES TERMINATED BY '\n'
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'hdfs://namenode:9000/result/ota/order_num_bydate';
CREATE EXTERNAL TABLE 'order_num_byprovince' (
  ' provincename' string,
  ' ordernum' int)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LINES TERMINATED BY '\n'
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.'TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'hdfs://namenode:9000/result/ota/order_num_byprovince';
```

（3）Hive添加UDTF。

```
[root@namenode soft]# cp /soft/hive-function-1.0.jar
/usr/local/hive-0.13.1-cdh5.3.6/lib/
hive> add jar /usr/local/hive-0.13.1-cdh5.3.6/lib/hive-function-1.0.jar;
```

（4）Hive计算。

未来一个月内每天的预定量如下。

```
insert overwrite table order_num_bydate
 select hotelid, concat('\'', hotelname, '\''), concat('\'', indate, '\''),
count(*) as num from hotelorder LATERAL VIEW split_date(arrivaldate,
departuredate, 'yyyy-MM-dd') tmp AS indate where indate >= '2016-12-01' and
indate <='2016-12-31' group by hotelid, hotelname, indate;
```

全国酒店预定量分布如下。

```
insert overwrite table order_num_byprovince
select concat('\'', provincename, '\''), count(*) as num from hotelorder
LATERAL VIEW split_date(arrivaldate, departuredate, 'yyyy-MM-dd') tmp AS
indate where indate = '2016-12-28' group by provincename;
```

### 11.7.4 配置 Sqoop

（1）创建 MySQL 表，在 Hive 的安装目录 bin 目录下执行命令./hive 进入 Hive Shell 环境，执行创建表命令，如下所示。

```
mysql> create database ota;
mysql> use ota;
Database changed
```

（2）未来一个月内每天的预定量。

```
CREATE TABLE 'order_num_bydate' (
  'hotelid' int NOT NULL,
  'hotelname' varchar(40) NOT NULL,
  'indate'  varchar(40) NOT NULL,
  'ordernum' int NOT NULL,
  PRIMARY KEY ('hotelid', 'indate')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

（3）全国酒店预定量分布。

```
CREATE TABLE 'order_num_byprovince' (
  'provincename' varchar(40) NOT NULL,
  'ordernum' int(11) NOT NULL,
  PRIMARY KEY ('provincename')
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

（4）在 MySQL 命令环境中进行授权。

```
GRANT ALL ON ota.* TO 'root'@'namenode' identified BY 'root';
```

（5）解压 Sqoop 至/usr/local，执行命令 tar -zxvf /soft/sqoop2-1.99.4-cdh5.3.6.tar.gz -C /usr/local/。如图 11-50 所示。

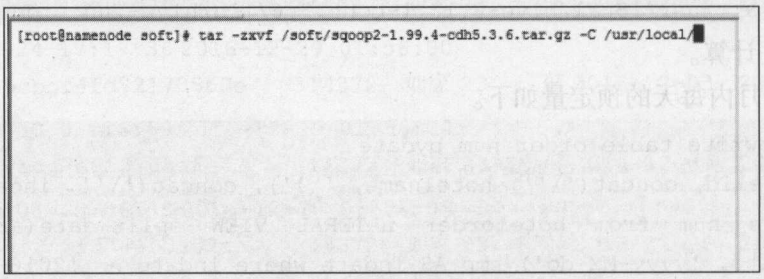



图 11-50 解压 Sqoop

（6）修改环境变量，执行命令 vim /etc/profile，增加如下内容。

```
export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
```

然后执行命令 source /etc/profile，对环境变量进行刷新。如图 11-51 所示。

```
    umask 022
fi

for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done

unset i
unset -f pathmunge

export JAVA_HOME=/usr/local/jdk1.7.0_79/
export PATH=$PATH:$JAVA_HOME/bin

export HADOOP_HOME=/usr/local/hadoop-2.5.0-cdh5.3.6
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HIVE_HOME=/usr/local/hive-0.13.1-cdh5.3.6
export PATH=$PATH:$HIVE_HOME/bin

export SQOOP_HOME=/usr/local/sqoop2-1.99.4-cdh5.3.6
export PATH=$PATH:$SQOOP_HOME/bin:$SQOOP_HOME/server/bin
"/etc/profile" 90L, 2170C                                    90,1          Bot
```

图 11-51 配置环境变量

（7）修改 ClassPath，在命令终端，执行命令 vim /usr/local/sqoop2-1.99.4-cdh5.3.6/server/conf/catalina.properties ，在文件中增加如下内容：

```
common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.
home}/lib,${catalina.home}/lib/*.jar,${catalina.home}/../lib/*.jar,/usr/
local/hadoop-2.5.0-cdh5.3.6/share/hadoop/common/*.jar,/usr/local/hadoop-
2.5.0-cdh5.3.6/share/hadoop/common/lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/
share/hadoop/hdfs/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/hdfs/
lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/*.jar,/
usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/lib/*.jar,/usr/local/
hadoop-2.5.0-cdh5.3.6/share/hadoop/yarn/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/
share/hadoop/yarn/lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/lib/*.jar
```

如图 11-52 所示。

```
#common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.home}/lib,${
catalina.home}/lib/*.jar,${catalina.home}/../lib/*.jar,/usr/lib/hadoop/*.jar,/usr/lib/
hadoop/lib/*.jar,/usr/lib/hadoop-hdfs/*.jar,/usr/lib/hadoop-hdfs/lib/*.jar,/usr/lib/ha
doop-mapreduce/*.jar,/usr/lib/hadoop-mapreduce/lib/*.jar,/usr/lib/hadoop-yarn/*.jar,/u
sr/lib/hadoop-yarn/lib/*.jar
#
#common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.home}/lib,${
catalina.home}/lib/*.jar,${catalina.home}/../lib/*.jar,/usr/lib/hadoop/*.jar,/usr/lib/
hadoop/lib/*.jar,/usr/lib/hadoop-hdfs/*.jar,/usr/lib/hadoop-hdfs/lib/*.jar,/usr/lib/ha
doop-mapreduce/*.jar,/usr/lib/hadoop-mapreduce/lib/*.jar,/usr/lib/hadoop-yarn/*.jar,/u
sr/lib/hadoop-yarn/lib/*.jar
common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.home}/lib,${c
atalina.home}/lib/*.jar,${catalina.home}/../lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6
/share/hadoop/common/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/common/lib/*.
jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/hdfs/*.jar,/usr/local/hadoop-2.5.0-c
dh5.3.6/share/hadoop/hdfs/lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapr
educe/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/lib/*.jar,/usr/loc
al/hadoop-2.5.0-cdh5.3.6/share/hadoop/yarn/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/shar
e/hadoop/yarn/lib/*.jar,/usr/local/hadoop-2.5.0-cdh5.3.6/lib/*.jar
                                                              59,1          62%
Connected to 139.129.226.141        SSH2 - aes128-cbc - hmac-md5  86x20
```

图 11-52 修改 ClassPath

（8）添加 Hadoop 配置文件，执行命令 vim /usr/local/sqoop2-1.99.4-cdh5.3.6/server/conf/sqoop.properties ，在文件中增加内容如下：

```
org.apache.sqoop.submission.engine.mapreduce.configuration.directory=/usr/local/hadoop-2.5.0-cdh5.3.6/etc/hadoop/。
```

如图 11-53 所示。

```
# Number of milliseconds for purge thread to sleep, by default one day
#org.apache.sqoop.submission.purge.sleep=

# Number of milliseconds for update thread to sleep, by default 5 minutes
#org.apache.sqoop.submission.update.sleep=

#
# Configuration for Mapreduce submission engine (applicable if it's configured)
#

# Hadoop configuration directory
org.apache.sqoop.submission.engine.mapreduce.configuration.directory=/usr/local/hadoop-2.5.0-cdh5.3
.6/etc/hadoop/

#
# Execution engine configuration
#
org.apache.sqoop.execution.engine=org.apache.sqoop.execution.mapreduce.MapreduceExecutionEngine

#
# Authentication configuration
#
#org.apache.sqoop.authentication.type=SIMPLE
#org.apache.sqoop.authentication.handler=org.apache.sqoop.security.SimpleAuthenticationHandler
#org.apache.sqoop.anonymous=true
#org.apache.sqoop.authentication.type=KERBEROS
-- INSERT --                                                    150,1         96%
```

图 11-53　添加 Hadoop 配置文件

（9）拷贝 MySQL 驱动。

```
cp /soft/mysql-connector-java-5.1.40.jar /usr/local/sqoop2-1.99.4-cdh5.3.6/lib/。
```

如图 11-54 所示。

```
[root@namenode ~]# cp /soft/mysql-connector-java-5.1.40.jar /usr/local/sqoop2-1.99.4-cdh5.3.6/lib/
```

图 11-54　拷贝 MySQL 驱动

（10）启动 Server。如图 11-55 所示。

```
sqoop.sh server start
```

```
[root@namenode ~]# sqoop.sh server start
```

图 11-55　启动 Server

（11）拷贝 MySQL 驱动至 hadoop 中，执行命令 cp /soft/mysql-connector-java-5.1.40.jar /usr/local/hadoop-2.5.0-cdh5.3.6/share/hadoop/common/lib/mysql-connector-java-5.1.40.jar，如图 11-56 所示。

```
[root@namenode ~]# cp /soft/mysql-connector-java-5.1.40.jar /usr/local/hadoop-2.5.0-cdh5.3.6/share/
hadoop/common/lib/mysql-connector-java-5.1.40.jar
```

图 11-56 拷贝 MySQL 驱动

### 11.7.5 从 HDFS 导出数据至 MySQL

（1）启动 Sqoop Client，在 Sqoop 安装目录的 bin 目录下，执行命令 sqoop.sh client，启动 Sqoop 客户端并进行相关设置及查看版本和连接。如图 11-57 所示。

```
[root@namenode bin]# ll
total 8
-rwxr--r-- 1 1106 592 3601 Jul 29  2015 sqoop.sh
-rwxr--r-- 1 1106 592 1361 Jul 29  2015 sqoop-sys.sh
[root@namenode bin]# sqoop.sh client
Sqoop home directory: /usr/local/sqoop2-1.99.4-cdh5.3.6
Sqoop Shell: Type 'help' or '\h' for help.

sqoop:000> show verison --all
The specified function "verison" is not recognized.
sqoop:000> set server --host namenode
Server is set successfully
sqoop:000> show version --all
client version:
  Sqoop 1.99.4-cdh5.3.6 source revision ed5e99c6207f3b5a673710934fc7bc85e34e11f3
  Compiled by jenkins on Tue Jul 28 15:04:37 PDT 2015
Exception has occurred during processing command
Exception: org.apache.sqoop.common.SqoopException Message: CLIENT_0000:An unknown error has occurre
d
sqoop:000> show connector
Exception has occurred during processing command
Exception: org.apache.sqoop.common.SqoopException Message: CLIENT_0000:An unknown error has occurre
d
sqoop:000>
```

图 11-57 启动 Sqoop Client

（2）未来一个月内每天的预定量。

创建 jdbc-connector

```
sqoop:000> create link -c 2
```

创建 HdfsConnector

```
sqoop:000> create link -c 1
```

创建 Job

```
sqoop:000> create job -f 6 -t 5
```

启动 Job

```
sqoop:000> start job -j 9 -s
```

（3）检查 MySQL 导出结果。

```
mysql> select * from order_num_bydate;
|  166843 | 全季      | 2016-12-13 |     1854 |
|  166843 | 全季      | 2016-12-14 |     1655 |
|  166843 | 全季      | 2016-12-15 |     1731 |
|  166843 | 全季      | 2016-12-16 |     1898 |
```

```
|  166843 | 全季      | 2016-12-17 |     2367 |
|  166843 | 全季      | 2016-12-18 |     2256 |
|  166843 | 全季      | 2016-12-19 |     2425 |
```

（4）全国酒店预定量分布。

创建 Job

```
sqoop:000> create job -f 6 -t 5
```

启动 Job

```
sqoop:000> start job -j 10 -s
```

检查 MySQL 导出结果

```
mysql> select * from order_num_byprovince;
| 上海         |      637 |
| 云南         |      123 |
| 内蒙古       |       33 |
| 北京         |      866 |
| 台湾         |        7 |
| 吉林         |       70 |
| 四川         |      477 |
| 天津         |       79 |
| 宁夏         |       69 |
| 安徽         |      130 |
| 山东         |      237 |
| 山西         |      165 |
```

## 11.8 数据展示

### 11.8.1 搭建 Web 开发环境

（1）集成 Tomcat。

首先选择“Window”→“Show View”→“Servers”。如图 11-58 所示。

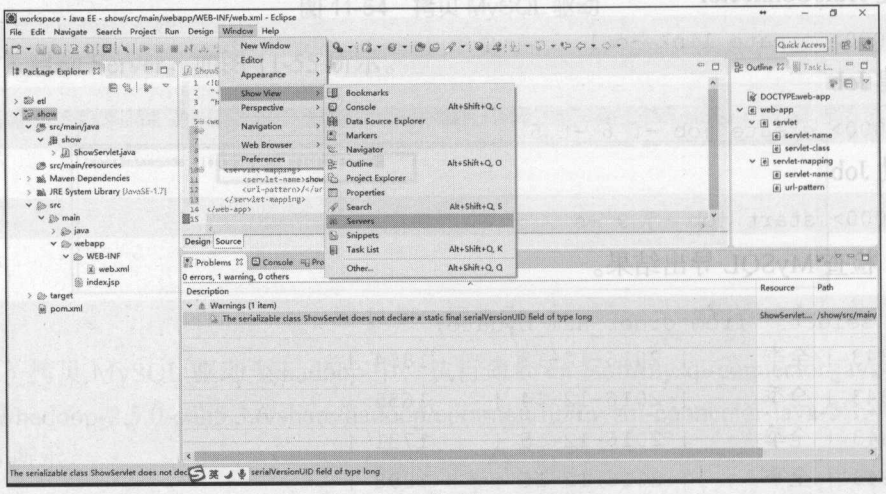



图 11-58　集成 Tomcat 1

在 Servers 视图窗口中，弹出“New Server”界面，选择“Tomcat v8.5 Server”，单击“Next”。如图 11-59 所示。

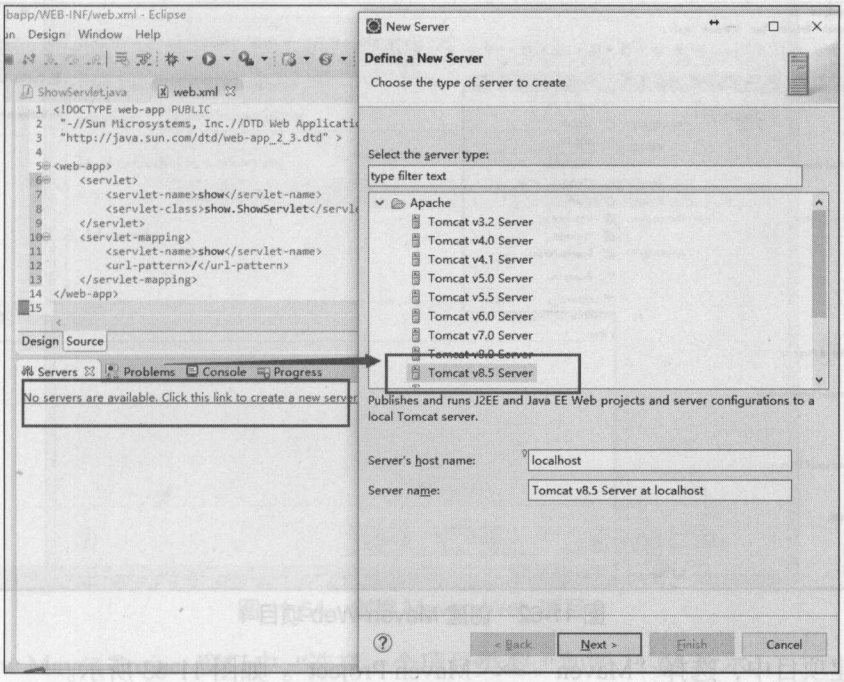



图 11-59 集成 Tomcat 2

在 Tomcat 配置界面，单击“Browse”，弹出浏览文件夹窗口，找到 Tomcat 的安装路径，并单击“确定”。如图 11-60 所示。

单击“Add”按钮，把 show 项目添加到 Server 上。如图 11-61 所示。

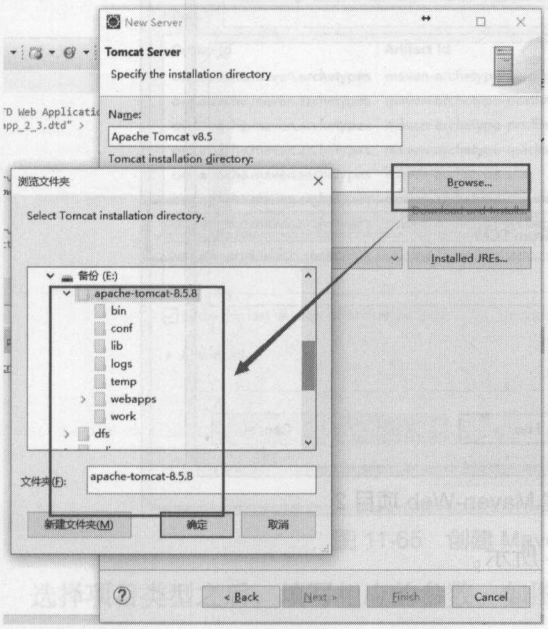



图 11-60 集成 Tomcat 3

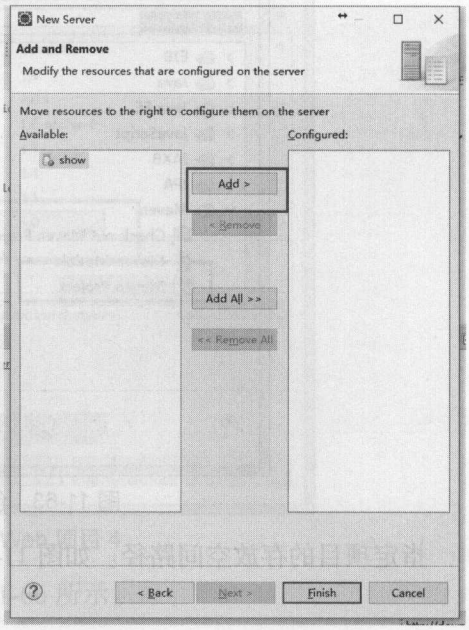



图 11-61 集成 Tomcat 4

（2）创建 Maven-Web 项目。

在项目浏览列表中，单击右键，选择“New”→“Project”。如图 11-62 所示。

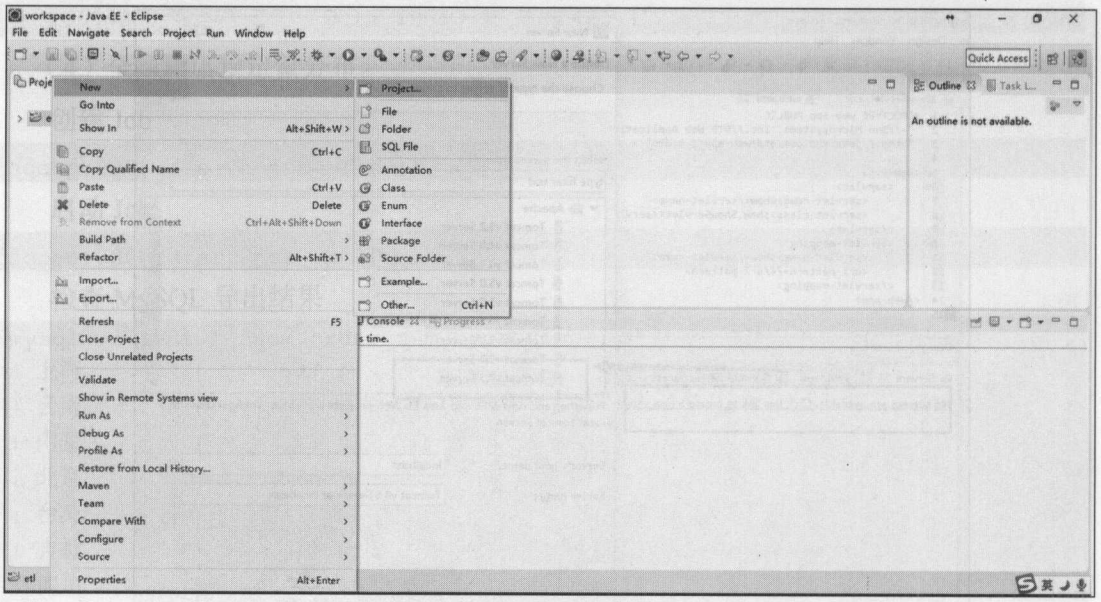



图 11-62　创建 Maven-Web 项目 1

在新建项目中，选择“Maven”→“Maven Project”。如图 11-63 所示。

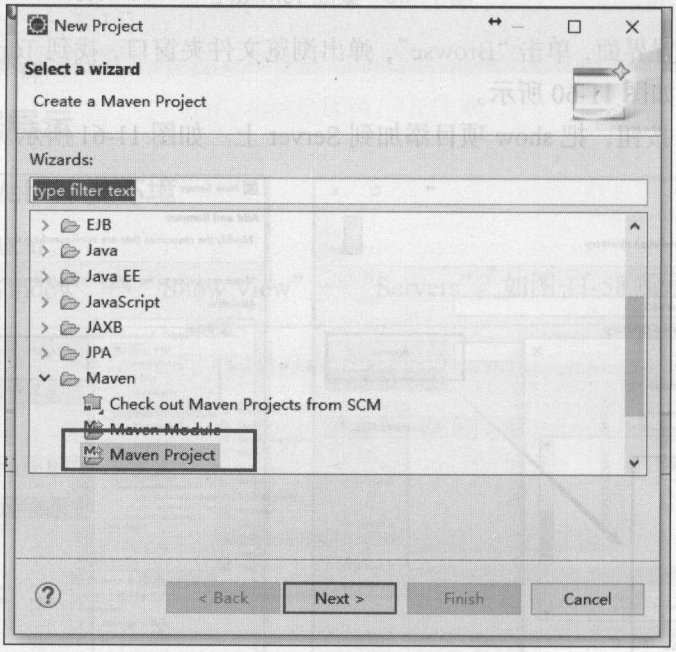



图 11-63　创建 Maven-Web 项目 2

指定项目的存放空间路径。如图 11-64 所示。

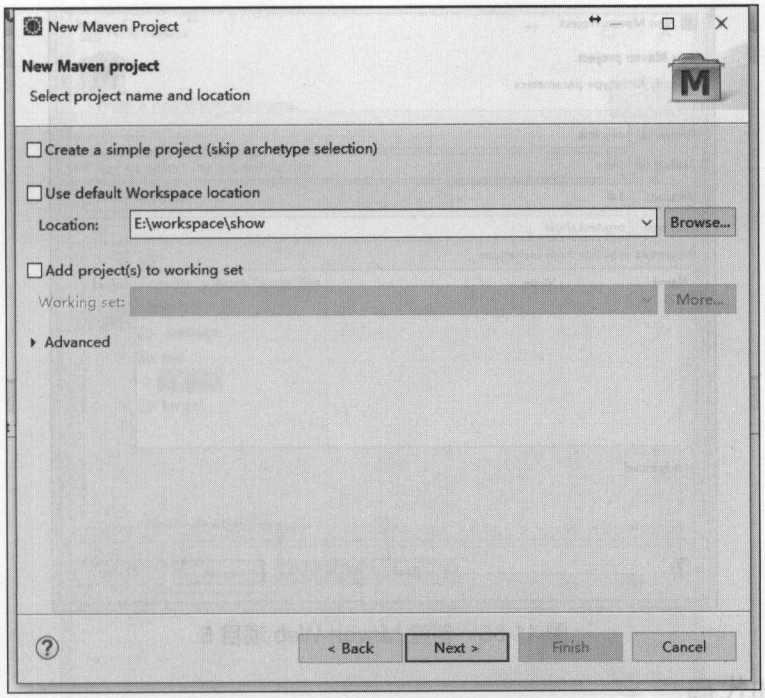



图 11-64 创建 Maven-Web 项目 3

在 New Maven Project 窗口中，选择一个项目类型。如图 11-65 所示。

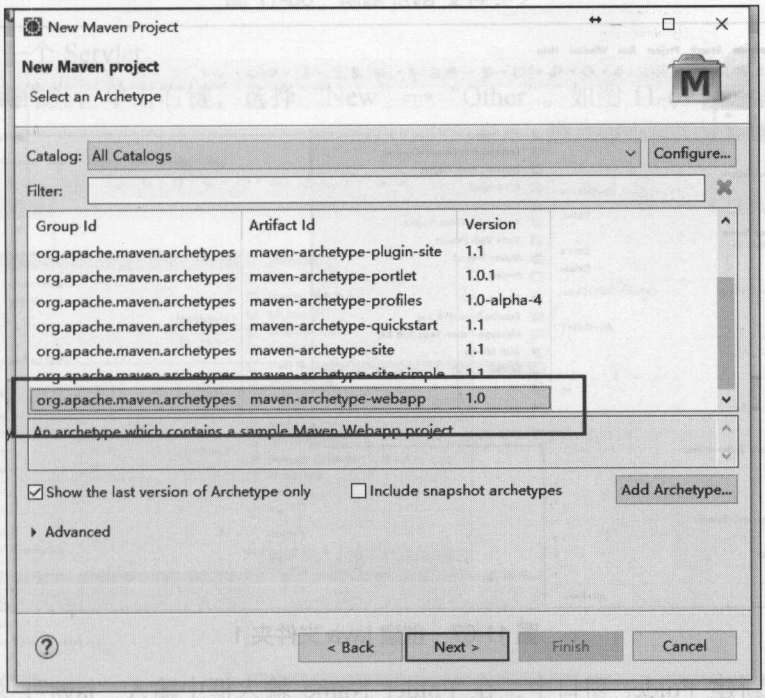



图 11-65 创建 Maven-Web 项目 4

选择项目类型之后，填写相应的参数。如图 11-66 所示。

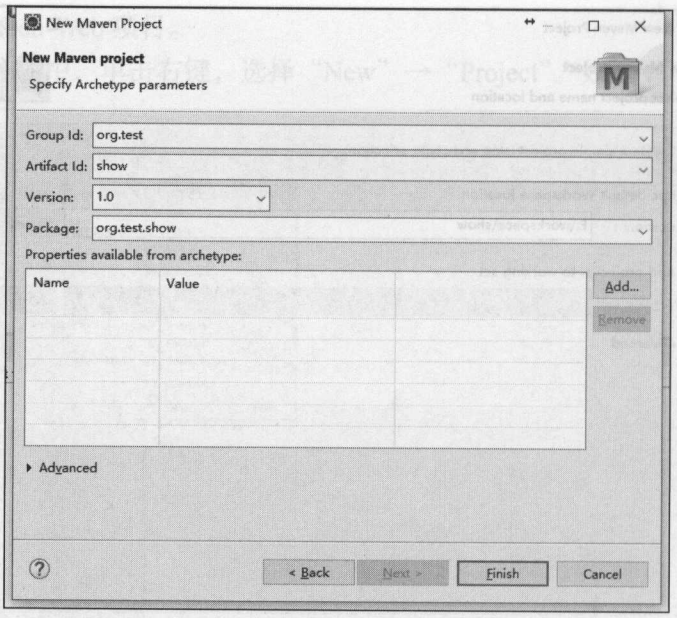



图 11-66　创建 Maven-Web 项目 5

## 11.8.2　添加代码

（1）创建 src/main/java 文件夹。

选中 main 文件夹，单击右键，选择“New”→“Folder”。如图 11-67 所示。

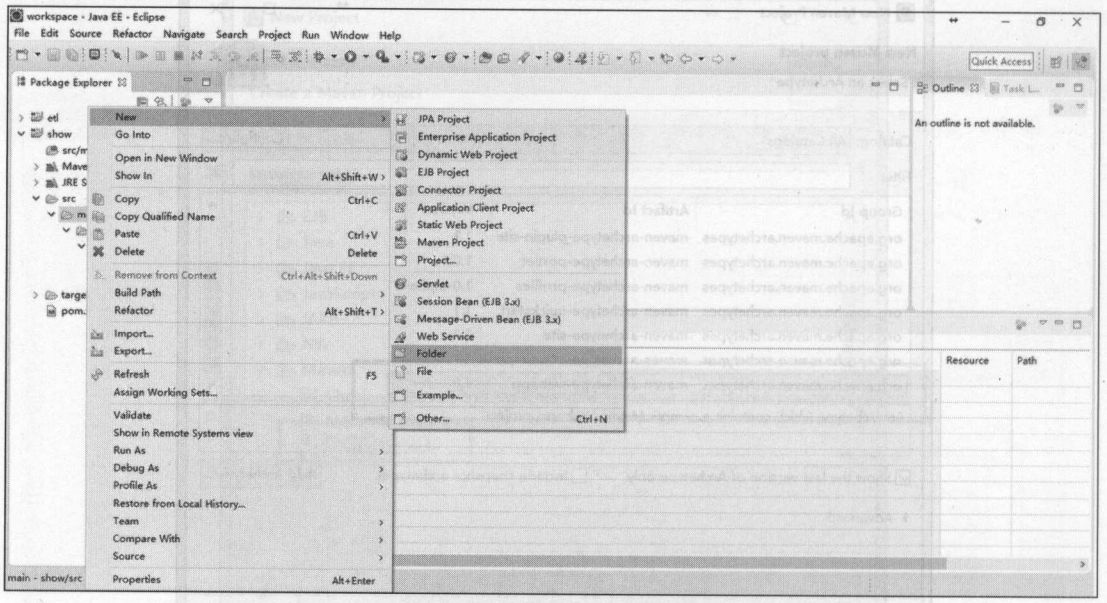



图 11-67　创建 java 文件夹 1

在弹出的新建 Folder 窗口中，在 Folder Name 输入框中输入“java”，然后左键单击“Finish”按钮。如图 11-68 所示。

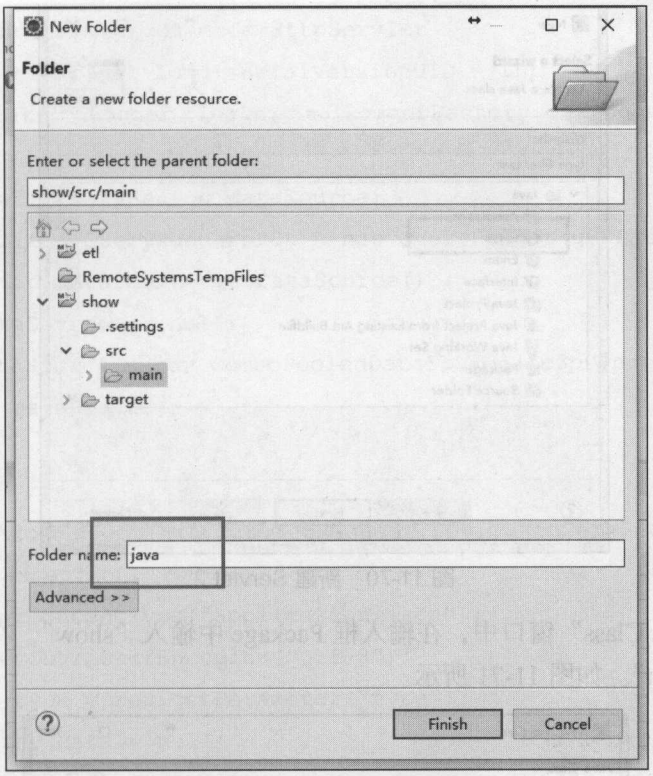



图 11-68 创建 java 文件夹 2

（2）新建一个 Servlet。

选中 src 文件夹，单击右键，选择“New”→“Other”。如图 11-69 所示。

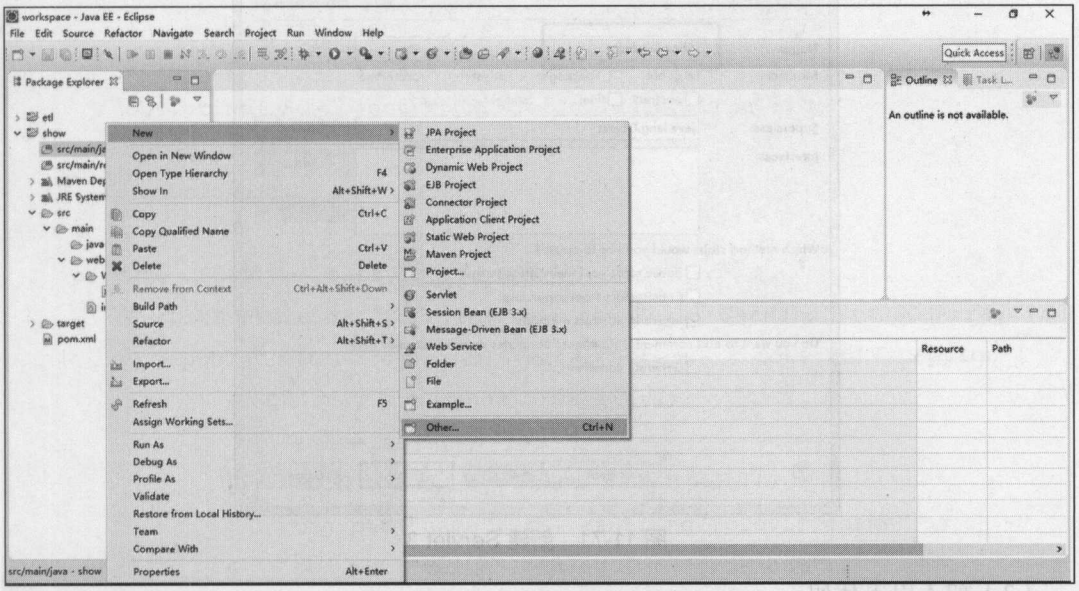



图 11-69 新建 Servlet 1

在弹出的 New 窗口中，选择“Java”→“Class”，然后选择“Next”。如图 11-70 所示。

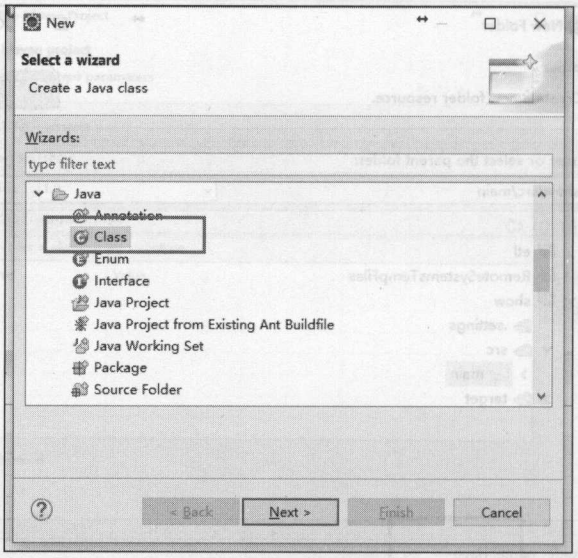



图 11-70　新建 Servlet 2

在“New Java Class”窗口中，在输入框 Package 中输入“show”，在 Name 输入框中，输入“ShowServlet”。如图 11-71 所示。

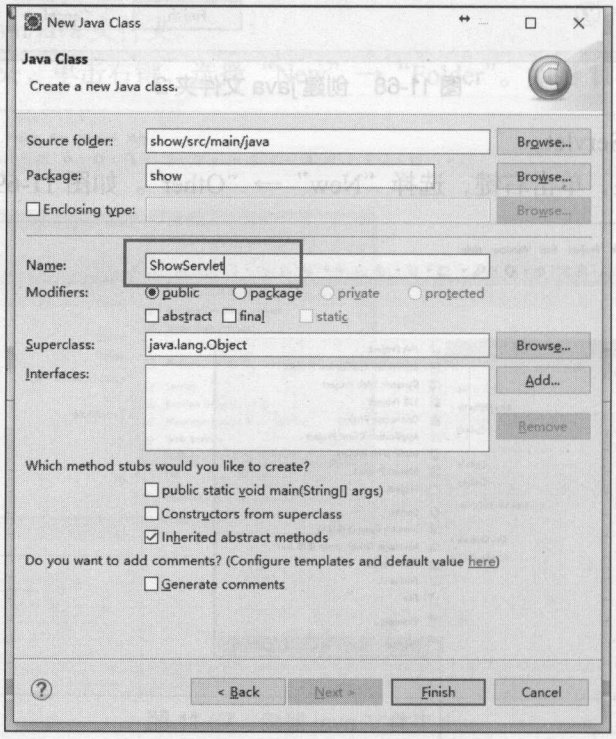



图 11-71　新建 Servlet 3

（3）加入以下代码。

```
/show/src/main/java/show/ShowServlet.java
package show;
```

```
public class ShowServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static Logger logger = LoggerFactory.getLogger(ShowServlet.
class);
    private static DataSource dataSource;
    protected final QueryRunner run = new QueryRunner(getDataSource());
    public static DataSource getDataSource() {
        if (dataSource == null)
            dataSource = new ComboPooledDataSource("c3p0.properties");
        return dataSource;
    }
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.setContentType("application/json");
        resp.setCharacterEncoding("utf-8");
        String type = req.getParameter("type");
        JSONArray data = null;
        if ("date".equals(type)) {
            data = getByDate();
        } else {
            data = getByProvince();
        }
        PrintWriter out = resp.getWriter();
        out.print(JSONObject.toJSONString(data, true));
        out.flush();
        out.close();
    }
    private JSONArray getByProvince() {
        ResultSetHandler<JSONArray> h = new ResultSetHandler<JSONArray>() {
            public JSONArray handle(ResultSet rs) throws SQLException {
                JSONArray arr = new JSONArray();
                while (rs.next()) {
                    JSONObject obj = new JSONObject();
                    obj.put("name", rs.getString("provincename"));
                    obj.put("value", rs.getInt("ordernum"));
                    arr.add(obj);
                }
                return arr;
```

```
        }
    };
    String sql = "SELECT * FROM order_num_byprovince";
    JSONArray list = null;
    try {
        list = run.query(sql, h);
    } catch (SQLException e) {
        logger.error("", e);
    }
    return list;
}
private JSONArray getByDate() {
    ResultSetHandler<JSONArray> h = new ResultSetHandler<JSONArray>() {
        public JSONArray handle(ResultSet rs) throws SQLException {
            JSONArray arr = new JSONArray();
            String tmpHotelName = null;
            List<Integer> list = new ArrayList<Integer>();
            while (rs.next()) {
                String hotelname = rs.getString("hotelname");
                if (tmpHotelName == null) {
                    list.add(rs.getInt("ordernum"));
                } else {
                    if (tmpHotelName.equals(hotelname)) {
                        list.add(rs.getInt("ordernum"));
                    } else {
                        JSONObject bean = new JSONObject();
                        List<Integer> tmpList = new ArrayList<Integer>();
                        tmpList.addAll(list);
                        bean.put("data", tmpList);
                        bean.put("name", tmpHotelName);
                        bean.put("type", "line");
                        bean.put("stack", "总量");
                        arr.add(bean);
                        list.clear();
                        list.add(rs.getInt("ordernum"));
                    }
                }
                tmpHotelName = hotelname;
            }
```

```
                if (!list.isEmpty()) {
                    JSONObject bean = new JSONObject();
                    List<Integer> tmpList = new ArrayList<Integer>();
                    tmpList.addAll(list);
                    bean.put("data", tmpList);
                    list.clear();
                    bean.put("name", tmpHotelName);
                    bean.put("type", "line");
                    bean.put("stack", "总量");
                    arr.add(bean);
                }
                return arr;
            }
        };
        String sql = "SELECT * FROM order_num_bydate order by hotelid, indate";
        JSONArray list = null;
        try {
            list = run.query(sql, h);
        } catch (SQLException e) {
            logger.error("", e);
        }
        return list;
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

（4）修改 pom.xml，参见源码。

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.test</groupId>
    <artifactId>show</artifactId>
```

```
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>show Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        ..................
    </dependencies>
    <properties>
        <maven-war-plugin.version>2.6</maven-war-plugin.version>
        <maven-compiler-plugin.version>3.5.1</maven-compiler-plugin.version>
        <java.version>1.7</java.version>
    </properties>
    <build>
        ..................
    </build>
</project>
```

（5）修改 src\main\webapp\WEB-INF\web.xml。

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>show</servlet-name>
        <servlet-class>show.ShowServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>show</servlet-name>
        <url-pattern>/json</url-pattern>
    </servlet-mapping>
</web-app>
```

（6）修改 index.jsp。

```
<html>
<title>展示页</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<script type="text/javascript" src="/show/js/jquery-1.9.1.min.js"></script>
<script type="text/javascript" src="/show/js/bootstrap.js"></script>
<script type="text/javascript" src="/show/js/echarts.min.js"></script>
```

```
<script type="text/javascript" src="/show/js/shine.js"></script>
<script type="text/javascript">
    // 为echarts对象加载数据
    $(document).ready(function() {
        $.ajax({
            url : "/show/json?type=date",
            async : false,
            type : "get",
            error : function(response) {
                alert("请求发送失败");
            },
            success : function(response) {
                drawLine(response);
            }
        });
        function drawLine(data) {
            var myChart = echarts.init(document.getElementById("line"), 'shine');
            myChart.setOption({
                title : {
                    text : '未来一个月内预定量变化趋势'
                },
                tooltip : {
                    trigger : 'axis'
                },
                legend: {
                   data:['汉庭','如家','七天','全季']
                },
                grid : {
                    left : '3%',
                    right : '4%',
                    bottom : '3%',
                    containLabel : true
                },
                toolbox : {
                    feature : {
                        saveAsImage : {}
                    }
                },
```

```
            xAxis : {
                type : 'category',
                boundaryGap : false,
                data  :  [  '12/1','12/2','12/3','12/4','12/5','12/6',
'12/7','12/8','12/9','12/10','12/11','12/12','12/13','12/14','12/15','12/
16','12/17','12/18','12/19','12/20','12/21','12/22','12/23','12/24','12/25',
'12/26','12/27','12/28','12/29','12/30' ]
            },
            yAxis : {
                type : 'value'
            },
            series : data
        });
    }
    // JSON
    $.getJSON('/show/js/china.json', function (data) {
        echarts.registerMap('china', data);
        drawMap(data);
    });
    var mapData;
    $.ajax({
        url : "/show/json?type=province",
        async : false,
        type : "get",
        error : function(response) {
            alert("请求发送失败");
        },
        success : function(response) {
            mapData = response;
        }
    });

    function drawMap(data) {
        var myChart = echarts.init(document.getElementById("map"), 'shine');
        myChart.setOption({
            title: {
                text: '全国酒店 12/28 预定量分布',
                subtext: '',
```

```
        left: 'center'
    },
    tooltip: {
        trigger: 'item'
    },
    legend: {
        orient: 'vertical',
        left: 'left',
        data:['全国']
    },
    visualMap: {
        min: 0,
        max: 2500,
        left: 'left',
        top: 'bottom',
        text: ['高','低'],           // 文本，默认为数值文本
        calculable: true
    },
    toolbox: {
        show: true,
        orient: 'vertical',
        left: 'right',
        top: 'center',
        feature: {
            dataView: {readOnly: false},
            restore: {},
            saveAsImage: {}
        }
    },
    series: [
        {
            name: '全国',
            type: 'map',
            mapType: 'china',
            roam: false,
            label: {
                normal: {
                    show: true
```

```
                        },
                        emphasis: {
                            show: true
                        }
                    },
                    data:mapData
                }
            ]
        });
    }
});
</script>
<body>
    <div id="line" style="height: 400px; width: 900px"></div>
    <div id="map" style="height: 400px; width: 900px"></div>
</body>
</html>
```

（7）添加/show/src/main/resources/log4j.properties。

```
log4j.rootLogger=INFO,stdout,R
# stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] %c{2}:%L %m%n
log4j.appender.stdout.Encoding=UTF-8

# rolling log file
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.maxFileSize=1GB
log4j.appender.R.maxBackupIndex=10
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %c (line %L)
%m%n
log4j.appender.R.File=logs/server.log
log4j.appender.R.Encoding=UTF-8
```

### 11.8.3 项目结构

项目结构如图 11-72 所示。

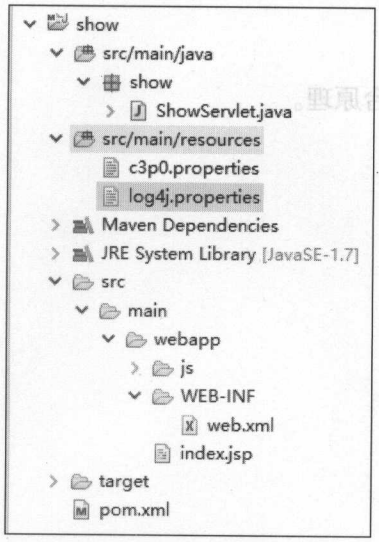



图 11-72 项目结构

### 11.8.4 启动 Tomcat

在 Server 视图窗口中，单击右键“Tomcat v8.5”，然后选择“Start”，启动 Tomcat 服务。如图 11-73 所示。

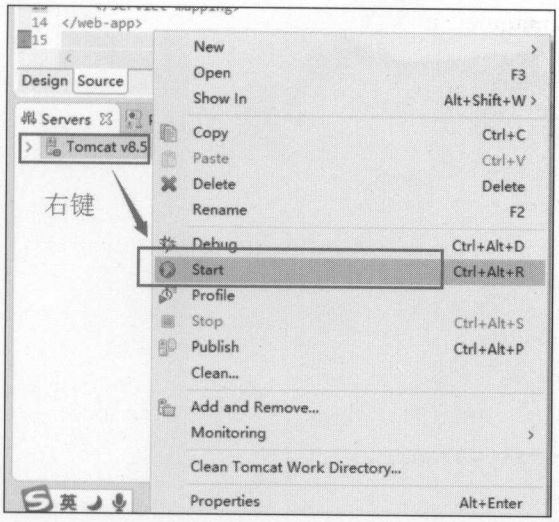



图 11-73 启动 Tomcat

### 11.8.5 访问 Web 页面

Tomcat 服务启动之后，在浏览器中输入网址 http://localhost:8080/show/。

## 本章小结

本章通过前面学习的知识完成一个利用大数据相关技术对 OTA 离线数据分析的应用，主要涉及的知识有 Hadoop 集群环境的搭建，Hive 工具的使用，MapReduce 对数据的处理，分析数据结果的可视化。

## 习题

1. 简述离线数据分析平台原理。
2. 简述数据收集技术。
3. 简述数据分析技术。
4. 简述数据展示技术。

扫一扫在线测

U-SaaS 课程 | 实验 | 题库

教育部产学合作协同育人项目成果教材
西普教育研究院 IT 前沿技术方向高校系列教材

## 如何用“U-SaaS实验”


## 什么是“U-SaaS开放实验云平台”

U-SaaS是依托虚拟化和OpenStack架构等技术自主研发的基于Internet的在线实验平台，旨在帮助院校低成本建设虚拟仿真实验教学中心。

**近千个在线实验案例**

虚拟实验　互动答疑　综合测评

**实战化技能演练平台**

标准赛制　专业命题　寓教于乐

**全方位数据统计分析**

课程分析　测评报告　能力评估

免/费/提/供 PPT等教学相关资料

人邮教育 www.ryjiaoyu.com

教材服务热线：010-81055256
反馈 / 投稿 / 推荐信箱：315@ptpress.com.cn
人民邮电出版社教育服务与资源下载社区：www.ryjiaoyu.com

0006902
西普教育 www.simpleware.com.cn
获取激活码

ISBN 978-7-115-45360-0

ISBN 978-7-115-45360-0
定价：49.80 元

美术编辑：董志桢
封面设计：秦延新